Catenation and Operand Specialization for Tcl Virtual Machine Performance

by

Benjamin Vitale

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Catenation and Operand Specialization for

Tcl Virtual Machine Performance

Benjamin Vitale

Master of Science

Graduate Department of Computer Science

University of Toronto

2004

We present techniques for eliminating dispatch overhead in a virtual machine interpreter using a lightweight just-in-time native-code compilation. In the context of the Tcl VM, we convert bytecodes to native code, by concatenating the native instructions used by the VM to implement each bytecode instruction. We thus eliminate the dispatch loop. Furthermore, immediate arguments of bytecode instructions are substituted into the native code using run-time specialization. Native code output from C compiler is not amenable to relocation by copying; fix-up of the code is required for correct execution. Resulting code size increase is apparently impractical. We evaluate performance using hardware performance counters and system simulation. Some benchmarks achieve up to 50% speedup, but roughly half slow down, or exhibit little change. Most slowdown is attributable to I-cache overflow due to increased code size, and increased compilation time. Larger I-caches broaden applicability of technique.

Acknowledgements

I am indebted to my supervisor, Tarek Abdelrahman; my colleagues in the Compilers and Systems reading groups at the University of Toronto; the Tcl community; my second reader Angela Demke Brown; my wife, Anne Simms, and my daughter, Julia Vitale.

I am also grateful for the support of Virtutech AB [22] and its Simics Academic License Program. We used their Simics product for whole-system simulation during our development and evaluation processes.

Contents

List of Figures

Chapter 1

Introduction

Many portable high level languages are implemented using virtual machines. Examples include UCSD Pascal, Smalltalk, Forth, Java, and also scripting languages Tcl, Perl, and Python. A compiler translates the language to code for some virtual machine, which then interprets the code. The virtual machine may provide intimate linkage with a high-level run-time system that is more sophisticated than a typical general purpose hardware machine — for example, it may provide garbage collection, object systems, and other services. However, one penalty of the virtual machine approach is lost performance. Because the virtual machine code is interpreted, it runs slower — sometimes much slower — than native code.

The efforts to address this performance problem can be divided into two main camps. Sophisticated techniques can make the process of interpretation faster. Alternatively, compilers can translate the virtual machine code into native code, and execute that directly. Just-in-time (JIT) compilers make this translation at run-time. Avoiding a separate, slow, static compilation step means that a JIT compiler can be a drop-in replacement for an interpreter, preserving the interactive work modalities of a virtual machine system, especially important for scripting.

Unfortunately, a JIT compiler is typically larger and more complex than the virtual machine itself. It is large because it involves much of the complexity of a traditional static compiler, and, in addition, must deal with the complexities of compiling, linking, and loading code in a system process which is already running. Furthermore, it must not have a large start-up time, and must generate code quickly, so that the user of an application does not observe substantial delay at the beginning of, or during, program execution. Finally, it must generate code whose performance exceeds the interpreted version by a margin that is large enough to amortize the cost of compilation. All these factors mean extensive effort is required to develop a JIT.

1.1 Contributions

Our main contribution is two light-weight compiler techniques for translating bytecode for a virtual machine interpreter into native code. We refer to the first technique as *catenation*, and the second as *operand specialization*. Catenation eliminates the overhead of instruction dispatch in the virtual machine, while specialization decreases the overhead of operand fetch.

We apply our technique to the widely-used Tcl scripting language. Tcl already has a system to compile scripts to bytecodes on the fly, and then interpret them. While this compiler improves performance significantly, many Tcl applications still demand more speed. A native compilation step is a natural solution. The programming effort required in our approach is substantially less than a full-blown JIT compiler, but can still improve performance.

We built a system which employs catenation and specialization in a native code compiler for the Tcl virtual machine. Our compiler plays the role of a JIT, but with lower cost. It is simpler to develop, eschewing expensive optimizations and reusing code from the interpreter, which also reduces semantic errors. It compiles very quickly, working with fixed-size native code templates, and precomputes as much information as possible about these templates to further reduce delays. This speed is important to retaining the very short interactive edit-run cycle that motivated a JIT in the first place, instead of static compilation. We also apply a form of specialization to the catenated code. Such code is uniquely suited to operand specialization, because catenation yields a distinct, non-generic sequence of native code for each virtual instruction in the source bytecode program.

A second contribution of our work is an implemetation technique to synthesize these templates directly from the code for the existing interpreter, suitably manipulated by the C compiler and various object-code post-processing steps. This approach decreases programmer effort and semantic errors by reusing code.

1.2 Overview of Results

We show that our techniques meet many of their goals, including simplicity, semantic correctness of interpretation, and high compilation speed. They successfully eliminate dispatch overhead, and, together with operand specialization, this yields fairly consistent reduction in instruction counts for a given workload. Our experimental evaluation using 520 benchmarks show that, for workloads that have small kernels, we reduce execution time, sometimes substantially (up to 60% improvement.)

However, the process of catenation necessarily leads to code expansion. On a modern computer architecture, catenation grows the working set of many benchmarks beyond the size of typical instruction caches. While moderate cache size increases improve the advantage of our techniques, unrealistically generous caches are required to accelerate all benchmarks. Still, we believe each of our techniques has promise as part of more sophisticated JIT compilers. We study six benchmarks in detail to understand the effect of our code-expanding transformation on instruction cache behavior and processor throughput.

1.3 Outline of Thesis

In the next Chapter, we give some background on Tcl, and discuss the performance problems inherent in its implementation. We briefly describe the bytecode compiler and virtual machine. We compare the various classical strategies for instruction dispatch in virtual machines. In Chapter 3, we give a detailed description of catenation and specialization, our lightweight compilation techniques for VM interpreters, and in Chapter 4, show how to implement the techniques efficiently. Chapter 4 also describes how to reuse code from the interpreter to yield the required native-code templates, including the many details that arise when code from the compiler is copied, and moved, in ways not intended by the compiler or the ABI (application binary interface.) Chapter 5 discusses our experiments and empirical results. We continue in Chapter 6 with some related work in the field of virtual machine implementation, and in our concluding Chapter 7, we summarize our findings and discuss how the work might be extended to realize a full-blown JIT compiler.

Chapter 2

Background

This chapter covers background material helpful in understanding the rest of this thesis. We give an overview of Scripting languages in general, and Tcl in particular, including its bytecode compiler and virtual machine. In this context, we discuss more generally stack machines and virtual machine dispatch strategies.

2.1 Scripting languages

Our language of interest is Tcl. Tcl is a *scripting* language with over 500,000 users [31]. Other popular scripting languages are Perl [42] and Python [21]. All three are implemented with virtual machines. Scripting languages are popular because they are expressive at a high-level, provide dynamically-typed variables, have rich built-in standard libraries and data types, and are extensible. This last point is particularly important—many contemporary programming tasks consist of reusing existing software components, and/or integrating existing large software systems. For example, interfacing a web serving application to a database requires exactly this kind of integration. But reuse and integration require *glue*, that is, small components to abstract (wrap) and connect the larger components in a system, customize and configure them, and drive them with logic. This is called *scripting.*

Another advantage of scripting is portability, with resulting advantages in ease of deployment. Since they are high level, and implemented with virtual machines, one version of a scripting language program ("script") can run on many different platforms or architectures without changes.

Scripting languages often increase programmer productivity [34]. Tcl programmers are said to require one tenth the time to create the same program in Tcl than in C, and the resulting program will have roughly one tenth as many lines [30]. For this reason, Tcl and other scripting languages are used for many tasks requiring rapid development, including "business logic" and other large system configuration and integration, and, especially, software prototyping. Sometimes, a prototype results in software of sufficiently high quality that there is no need to re-implement it in a more traditional language. There are many very large production Tcl programs that either grew out of such prototypes, or that otherwise use Tcl for business logic, integration, configuration, embedded scripting, or some other reason. deployed [31]. These include AOL's Digital City, which uses Tcl for dynamic web content. Many companies use Tcl for automation of testing, including Cisco, to test its routers; Cygnus, to run regression tests on the GNU debugger (`gdb`) and other developer tools; and NBC, to automate digital video in its broadcast studios [28].

Performance is the main obstacle to creating entire applications in scripting languages. A secondary problem is structure. Large software projects demand some structure, or else they become unmaintainable. By permitting and even encouraging unstructured software architecture, scripting code can be inappropriate for large systems. Although most scripting languages provide facilities to create highly structured programs, if desired, performance is not easily improved, because the virtual machine is slow. Many Tcl users have said speed is the biggest obstacle to using Tcl for more applications [29].

2.2 Improving Performance

The typical approach to improve performance is to recode critical parts of the program in C, and link them with the interpreter, so they can be called from scripts. This is effective, but loses some of the advantages of scripting—portability, deploy-ability, and programmer productivity. The scripting languages have turned out to be full-featured general purpose languages, and

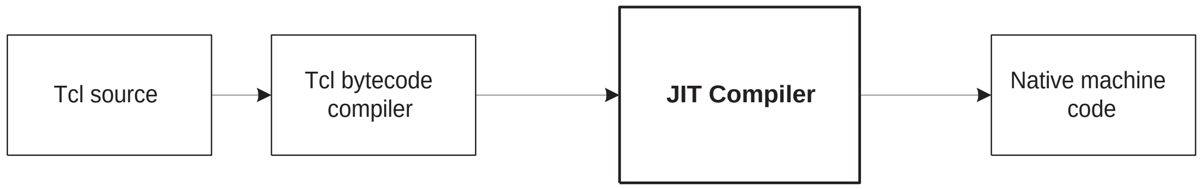


Figure 2.1: JIT compiler will use the output of the existing bytecode compiler

their users want to use them as such. Ideally, they could use the language directly, but extract better performance from the code.

One obvious way to improve performance is to compile to native code, instead of a virtual machine interpreter. It is possible to statically compile offline, and deploy the application already compiled. This is useful, but we strongly believe that JIT techniques, which compile only at run-time, are most appropriate for scripting languages. First of all, Tcl and other scripting languages have a dynamic nature, including many constructs, such as `eval`, which can create and invoke code on the fly. If such code is to execute natively, run-time compilation is required. Second, one of the key reasons for the high programmer productivity observed with scripting languages is the short edit-run cycle. This is in contrast to a longer edit-compile-debug-run cycle in traditional languages. Compiling automatically at run-time, when the script is run with its interpreter, retains the shorter edit-run cycle. The run-time JIT overhead is spread over execution time, because each program procedure is compiled as needed before execution, thus preserving rapid start-up time. Finally, scripting languages' high deployability is compromised if separate binaries must be released for every target architecture. Instead, strong portability is retained when the virtual machine on the target system is responsible for the native-code compilation step.

A native JIT compiler might start from the high level source, or it might start from the bytecode emitted by the existing bytecode compiler. The existing source-to-bytecode compiler provides a suitable front-end for the JIT. Furthermore, some Tcl modules are distributed only as bytecodes. So our system starts with these to produce native code, as shown in Figure 2.1.

Our techniques occupy a new design point in the continuum between simple interpretation and optimizing JIT compilers. As we shall discuss in Section 2.6, there are many other points in between, representing varying sophistication in interpretation, optimizing and dynamic JIT compilation, and multi-mode execution. Our technique lies just at the boundary between interpretation and native code execution. We execute native code, but it is substantially the same native code executed during traditional interpretation.

2.3 Language Fundamentals of Tcl

Tcl was invented in 1988 by Dr˙ John Ousterhout, then a professor at U. C. Berkeley. It was originally conceived as an "embeddable" command language. Many large computer software systems require some sort of "command" or "scripting" language, and these often end up being idiosyncratic, and mutually incompatible. The idea of Tcl was to provide a re-usable language component that could easily be integrated into existing applications, and connected to their existing libraries.

As such, Tcl's main strengths are that it is very easy to integrate into large existing applications, and very good as a "glue" language to bind together two (or more) systems with disparate APIs. However, as it matured, many of its hundreds of thousands [31] of users came to use Tcl as a programming language in its own right. As such, there exists some demand to increase the run time performance of Tcl. Because of its popularity in the embedded systems and digital design communities, there have even been attempts to execute Tcl bytecodes in FPGA hardware [40] to achieve the ultimate performance. But most users will not have access to special-purpose hardware, and require a software solution; native-code compilation is a natural fit. Before we present our ideas on how to do this, we give some background on Tcl syntax, philosophy, implementation, and performance.

Tcl is, by design, very easy to extend Tcl with C code, and this ameliorates some performance constraints. This ease has led to a wide variety of extensions available for Tcl, which provide access to many existing C libraries in a way that promotes automation, code re-use, and high performance. For instance, the powerful `Expect` extension [19] allows programs with interactive text user interfaces to be scripted by watching for patterns from and sending commands to a pseudo-tty (pty) connected to the application. It has found wide application in the software testing domain, forming the basis for Cisco Systems's 1 million line IOS test suite, and

```
puts "Hello World"
```

Figure 2.2: Canonical first program in Tcl

```
 1 # compute n!, i.e. factorial function
 2
 3 proc factorial {n} {
 4     set fact 1
 5     while {$n > 1} {
 6         set fact [expr {$fact * $n}]
 7         incr n -1
 8     }
 9     return $fact
10 }
```

Figure 2.3: Tcl proc to compute factorial function

the DejaGNUs test suite used by the GNU C compiler `gcc` and debugger `gdb`.

Another popular Tcl extension is `Tk`, a high-level binding onto `Xlib`, the client library for the MIT X11 Window GUI. It enables creating graphical user interfaces in just a few lines of Tcl code. Tcl and Tk both run on Microsoft Windows and MacOS, in addition to almost all Unix variants.

The Tcl language is used to script the primitives and objects in these various extension systems, and create interfaces between them. In the next section, we provide a brief overview of the core language itself.

2.3.1 Tcl Fundamentals

Figure 2.2 shows a simple program to print the string `Hello World` to standard output. The `puts` command takes an argument and sends it to standard output. The quotes around the string are necessary because the string contains a space, which would otherwise indicate multiple arguments to the command. That would be an error, because `puts` takes only one argument.

Figure 2.3 is a slightly more interesting program, to compute the `factorial` function. It demonstrates most of the other the syntactic elements of Tcl. Line 1 begins with a `#` character, which indicates a comment – anything up until the end of the line is ignored. Line 3 begins a `proc` – that is, a procedure or function. A `proc` definition needs three things: a name (`factorial`); a list of formal parameters ({`n`}); and a body (Lines 4 - 10.) The body is just regular Tcl code, which, while spread over multiple lines, is grouped together into a single argument by the {} characters.

Line 4 is the first line of the body of the `proc`; it contains a command. A command in Tcl is similar to one in the Unix Bourne shell [16]—it consists of the name of the command, `set`, and zero or more arguments. In this case, there are two arguments—`fact` and `1`. The command, and each of the arguments, are separated by white space. The command, the arguments, and indeed, everything else in Tcl, are simple strings. The semantics of Tcl are that *everything is a string*, and until Tcl 8.0 the internal implementation actually worked entirely with strings [18]. This command/arguments format is the extent of Tcl's syntax; there is no grammar.

The `set` command on Line 4 sets the value of a local variable, named `fact`, to 1. There are two kinds of scope in Tcl, *global* and *local*. Variables in procedures are local by default unless declared otherwise. There is no nested or block scope, although *namespaces*, while semantically much like globals, allow variables to be grouped in separate naming scopes.

Line 5 begins a `while` command; `while` is a control flow construct, as in other languages. However, in Tcl, its syntax is a command like any other. It takes two arguments, a condition and a body of code. It repeatedly evaluates the condition, and, while it is true, evaluates the body of code. The code is a just a string. Because the string contains white space and other special characters, it is grouped into a single argument using curly braces.

The condition in this example is `$n > 1`. The `$` in `$n` signifies variable dereference. That is, "value contained in the variable named `n`." However, there isn't any dereference occurring; everything in Tcl is a string, and all the semantics involve simple *substitutions*. The string `$n > 1` becomes, e.g., `5 > 1`, if the value of the variable `n` is `5`. Then, that new string is evaluated.

The substitution-oriented parsing of Tcl consists of a few basic elements:

- {} suppresses all substitution, and groups a string containing white space into a single string, instead of splitting it up into separate strings.

- \ quotes a special character. E.g. `puts "Here is a curly brace:  \{"`

- `$` substitutes the name of a variable with its value

- `""` are like {}, in that they group strings containing white space. However, it does allow for all other kinds of substitution. e.g.

  `puts "The value of the variable x is $x"`

- newlines and semicolons end statements. e.g. `puts "Hello"; puts "World"`

- [] essentially causes function call. The string inside the [] is evaluated as a Tcl script, and the result of the script is substituted for the [`script`]. We see this syntax on Lines 6 and 12 of our example in Figure 2.3.

Line 6 uses the command `expr`. In a more conventional programming language, this line might be written `fact := fact * n`. But Tcl's native semantics don't include any special provision for arithmetic! Instead, math is implemented with only commands and arguments. The `expr` command knows how to do math—that is, to evaluate arithmetic expressions. Its evaluator also does a second round of substitution, using all the normal Tcl parser substitution rules.

On Line 7, the `incr` command takes two arguments, the name of a variable and an integer. It increments the value of the variable by the integer. If either of its arguments do not meet these requirements, the interpreter generates an error.

On Line 9, the `return` statement stops execution of the procedure, and, optionally, returns a result to the caller.

Arithmetic in loops can be quite slow in Tcl, because of the need to continuously parse strings into numbers, and format the results of computations back into strings. While there exist Tcl extensions to perform advanced arithmetic (e.g. on arbitrary precision values) in C, the problem remains that if the original Tcl language is used, conversion to and from strings is required on each invocation of extension routines from Tcl. In the next section, we describe how Tcl performance issues are not confined to this one problem, and introduce a remedy.

2.3.2 The Need for Better Tcl Performance

Earlier, we described Tk, a Tcl extension for creating graphical user interfaces. While some Tk programs are not performance sensitive, others require hundreds or thousands of Tcl commands to be executed on every movement of the mouse pointer. If these commands are interpreted too slowly, the GUI becomes unusable. By compiling these to native code, the responsiveness of the GUI will increase.

Extending Tcl or writing Tcl programs using C creates at least two obstacles. First, even very carefully written C programs are less portable than Tcl. Furthermore, installing them requires that a user have a compiler, or install a binary library. So C extensions potentially decrease the deploy-ability of Tcl applications. Second, because C is a much lower level language, programmers take much more time to write C programs than Tcl. Tcl programs are typically ten times shorter, with a commensurate decrease in programmer effort [32].

Therefore, some demand for better performance of Tcl itself remains. In 1995, Brian Lewis, in the Tcl group at Sun Microsystems, created a “bytecode” compiler for Tcl version 8.0 [18]. This version was compatible with the semantics of earlier versions, but ran much faster — as much as 10 times faster. The compiler re-uses some of the Tcl compilation ideas in the work of Adam Sah, a student of Ousterhout’s at Berkeley, which we discuss with other Related Work in Chapter 6.

Until recently, little has been done to improve Lewis’s bytecode compiler, or exploit the opportunities it created. There is now some interest by the Tcl Core Team in improving its performance, but not compiling to native code, as we will do. Our native code compiler can allow more code written in Tcl, with less in C, thus improving programmer productivity.

In Tcl versions before 8.0, the Tcl interpreter implementation followed these rules as described above. In Tcl 8.0 and later, the semantics of the interpreter and compiler are consistent with these rules. However, the internal implementation works differently, for the sake of performance. The implementation is not documented elsewhere, except in the code itself. We provide some tutorial documentation here, since it is the foundation on which we build our later work.

2.4 The Tcl 8.0 bytecode compiler

There are two serious performance problems with the Tcl interpreter before version 8.0. Implementing *everything is a string* semantics meant a great deal of copying data, and a lot of

```
1  set x 6
2  set y 7
3  set z [expr {$x * $y}]
4  puts "$x * $y = $z"
```

Figure 2.4: Objects with numeric values and cached string representation

extra work as strings are parsed and re-parsed, for tokenizing and substitutions. Furthermore, arithmetic requires lots of extra parsing. For example, the argument to the `expr` command is a string. Evaluation of `expr {7 * 11 * 13}` requires a C `sscanf` of the ASCII strings `7`, `11`, and `13`, before performing the multiplication, and then an `sprintf` of the result to another ASCII string, `1001`. Evaluating a loop body or a new procedure meant parsing it repeatedly. Tcl 8.0 addresses these performance problems with two techniques:

- a reference-counted *object* system, which, for every string in the system, attempts to cache a typed value, such as an integer. This is known as a *dual-ported* object system.
- a *bytecode* compiler and virtual machine interpreter, which avoids repeated re-parsing by compiling each proc or script into a series of bytecode instructions.

Note that the object system only *caches* special typed values. Because of Tcl's *everything is a string* semantics, and a large body of C code interfacing with the Tcl interpreter (including the interpreter itself, and its library), each object must always be able to supply a string representation of itself on demand. This is why the objects are called dual-ported: each one is a string and a cached value.

For example, consider the code in Figure 2.4. In Lines 1, 2, and 3, the objects stored in the variables `x`, `y`, and `z` don't need any string representation. They are stored as integers, and do not need to translate from or to strings. However, in Line 4, the `puts` command demands a string argument, and so the string version of the objects is generated. This string is cached, and thus will not need to be re-generated, unless it is invalidated by assigning a new value to the variable.

There are object types for integers, booleans, floats, Unicode strings (to support international character sets), procedure bodies, and several other internal objects. Objects are reference counted, and automatically deleted when their reference count goes to zero. Objects have *copy-on-write* semantics, so they can be shared among many uses. A copy is made only when a shared object is changed. This avoids most copying for read-only data structures, and is particularly important for large structures.

The virtual machine is defined and implemented as a *stack machine*, as we elaborate in the next subsections. It is implemented in the C programming language. It is bundled together as part of the same program as the Tcl parser, interpreter, compiler, and runtime, and is thus a drop-in replacement for the non-compiling interpreter in all previous Tcl versions. We will maintain this transparent deployment strategy in our native-code compiler.

2.4.1 Stack Machines

The Tcl VM is a *stack machine*, by which we we mean that the opcodes must refer to data on a virtual stack of objects. There are no registers. Historically, most bytecoded virtual machines use a stack. There appear to be two main reasons for this. Most importantly, most instructions' operands are implicitly the top one or more elements of the stack. No register numbers need to be stored in the operands of the instruction. Thus the instructions are very compact —often as little as one byte (hence the name, *bytecode*). This saves storage and memory bandwidth, but can also directly improve execution time because instruction decode is faster [7]. Secondly, it is easy to write compilers for stack machines, because compilation processes (e.g. recursive descent parsing) often naturally lend themselves to a post-order translation of the source code expressions or parse tree. Finally, the stack machine is said to be architecturally neutral, because a register machine would have to choose some finite number of registers, which would likely be different from the number of registers on a real machine. We do not find this latter argument convincing, but the other reasons are persuasive.

The cost of a stack machine is limited expressive power. While the stack may be many objects deep, each instruction can only access the topmost items. In some circumstances, extra operations are necessary simply to move items around on the stack. Register machines do not have this problem. Furthermore, especially in virtual machines, the stack operations consume a small amount of extra time, to maintain the stack pointer. Finally, and perhaps most important

```
Command 1: set fact 1
      0   push1 0                # "1"
      2   storeScalar1 1         # var "fact"
      4   pop

Command 2: while {$n > 1} {
      5   loadScalar1 0          # var "n"
      7   push1 0                # "1"
      9   gt
     10   jumpFalse1 16          # pc 26

Command 3: set fact [expr {$fact * $n}]
     12   loadScalar1 1          # var "fact"
     14   loadScalar1 0          # var "n"
     16   mult
     17   storeScalar1 1         # var "fact"
     19   pop

Command 4: incr n -1
     20   incrScalar1Imm 0 -1
     23   pop
     24   jump1 -19              # pc 5
     26   push1 1                # ""
     28   pop

Command 5: return $fact
     29   loadScalar1 1          # var "fact"
     31   done
```

Figure 2.5: Tcl bytecode for factorial function in Figure 2.3

for JIT compilers, the stack code can obscure the flow of data in the program, and thus the semantics of the code [17]. This happens because the depth of a value on the stack changes as new values are pushed and popped, and is bound to the temporal ordering of instructions which may actually be independent of the value. This makes it difficult to determine which transformations are profitable, or even permissible, thus hampering optimization. Usually, the stack code must be converted to a three-address register scheme before any optimization. We don't do any optimizations in our current design, but discuss them along with other future work, at the end of the thesis.

2.4.2 Examining Tcl Bytecodes

To better understand the bytecode compiler, let's consider Figure 2.5, the bytecode produced for the factorial function in Figure 2.3.

```
puts "The factorial of 5 is [factorial 5]"
 Command 1: puts "The factor..."
  0   push1 0        # "puts"
  2   push1 1        # "The factorial ..."
 Command 2: factorial 5
  4   push1 2        # "factorial"
  6   push1 3        # "5"
  8   invokeStk1 2
 10   concat1 2
 12   invokeStk1 2
 14   done
```

Figure 2.6: Bytecode for Tcl source (shown) to exercise factorial function in Figure 2.3. See text for description. The comments in the code show the literal strings indicated by the small integer arguments to the `push` opcode, which are indices into the *literal table.*

The bytecode compiler compiles any Tcl script into a series of 97 different opcodes. Many of the opcodes have one or more operands, and also can pop data from or push data to a stack. Most operands, and the contents of the stack, are both in the form of Tcl objects. The operands can refer to a table of literal objects built by the compiler. For instance, in the `factorial` example, the object for the constant 1 is literal 0. When the compiler is able to clearly discern a local variable, it generates code to access that variable as a slot in an array of local Tcl objects. This is a big performance advantage, since otherwise the variable must be looked up by its string name in a hash table — on *every* access. The language does allow for dynamic variable names, which are difficult or impossible for the compiler to ascertain. These can refer either to indexed variables the compiler did identify in simpler code, or to new variables unseen during compilation. In either case, the virtual machine can resort to the slower hash lookup for correct execution. Since most variables are local, this is an important optimization.

The compiler compiles each proc the first time it is executed. No optimization is performed.

Now let's look at some less idealized code, to get a sense for the mix of low- and high-level bytecodes in the Tcl virtual machine.

In Figure 2.4.2 we see two important execution modalities making use of the opcode `invokeStk1`. This instruction takes a command and arguments, from separate consecutive items on the top of the stack, and invokes the command with the arguments. This compiled code is really just undertaking interpretation. Worse, if the compiler encounters part of a script which it lacks patterns to compile, it stuffs the whole thing in a string object, pushes it on top of the stack, and emits the `evalStk` opcode, which has to then parse, compile and execute the script at run-time – a *slow* interpreter.

`invokeStk1` is used here on Line 8, to invoke a procedure call. Many Tcl primitives, such as `set`, `expr`, and `while`, are compiled into lower-level bytecode. But others, such as `puts`, and calls to user-defined procs like `factorial`, use `invokeStk1`. From the command object on the stack, `invokeStk1` extracts the address of a C callback function, and calls it. The function has a signature like this:

```
int Tcl_PutsObjCmd (ClientData dummy, Tcl_Interp *interp, int objc, Tcl_Obj *const objv []);
```

A `ClientData` is just a `void *`, and is used as a *closure* in the case where one callback will be used for many Tcl procs. We'll see an example of this in the next paragraph. The `interp` is a pointer to the current Tcl interpreter, which maintains all the state for a running script, including a stack of call frames, all global variables, all defined procedures, etc. There can be multiple interpreters active in the same program, and, these can run concurrently in one C thread, or each interpreter can have its own thread; `objc` is a count of the number of arguments for this command, and objv is a vector of the arguments.

In the case of a primitive, such as `puts` or a library extension, the callback can directly implement the command. On the other hand, for all user-defined Tcl procs, a single C function is called, and its `ClientData` points to a `Proc` structure, which contains the definition of the proc. This includes the names and number of formal parameters and other information, and also the Tcl script body of the proc as an ASCII string, and, if it has been compiled, a `ByteCode` object. The C function `TclObjInterpProc()` creates a call frame, pushes it on the interpreter's stack of call frames, copies all the function arguments to the formal parameters, sets up the execution environment—including building an array to hold the local variables compiled into slots by the bytecode compiler, and then recursively calls the virtual machine interpreter.

```
#define Tcl_IncrRefCount(objPtr) \
    ++(objPtr)->refCount

#define PUSH_OBJECT(objPtr) \
    Tcl_IncrRefCount (*(++tosPtr) = (objPtr))

#define POP_OBJECT() (*tosPtr--)

for (;;) {
    switch (*pc) {
        case INST_DUP:
            objPtr = TOS; /* top of stack */
            PUSH_OBJECT (objPtr);
            pc++;
            continue;

        case INST_PUSH:
            ...
```

Figure 2.7: Sketch of code implementing Tcl 8 virtual machine dispatch loop

2.5 The Tcl 8 Virtual machine

The bytecode instructions output by the Tcl 8 bytecode compiler are executed using interpretation by the Tcl 8 virtual machine. The virtual machine maintains a virtual program counter, or `pc`, which points into the currently executing bytecode. Then, in an infinite loop, it executes a giant `switch` statement on the next opcode, as shown in Figure 2.7. The opcode does its work, likely manipulating program state and the virtual machine stack. Then, it increments the program counter by the number of bytes used by that instruction. The opcode itself always takes 1 byte, and then there are 0, 1, or 2 immediate arguments. These arguments may each be 1 or 4 bytes long. The opcode knows its total length, and it is responsible for incrementing `pc` appropriately. Finally, the loop runs again.

The loop exits on a `done` opcode, or in any of several error conditions. The machine also maintains the object stack, which consists of a finite size array of pointers to objects, and a stack pointer, which indicates the top of the stack. The machine requires that any bytecode submitted for interpretation declares the maximum depth stack it will require. This depth is calculated by the bytecode compiler, since it is used to allocated the stack in the existing interpreter. This is determined by the compiler when it creates the bytecode object.

Procedure calls in the virtual machine are implemented using one of the `invoke_` or `eval_` opcodes. These opcodes invoke Tcl interpreter functions at the C-level that build Tcl stack frames, invoke dynamic *traces*, and ultimately make a recursive native procedure call to the virtual machine itself, which is, of course, re-entrant. The procedure call process may also require resolving the name of the Tcl procedure into a Tcl bytecode object, or native C function, but usually the result of this resolution is cached after the first execution.

There have been various attempts at compiling Tcl, which we discuss in Chapter 6. Our technique lies somewhere between JIT compiling and interpreting bytecode. Since we don't translate the code to generate newly-synthesized native code, we feel we're best compared to the various instruction dispatch mechanisms for virtual machine interpreters.

The virtual machine interpreter executes the bytecode instructions one after the other. After fetching each bytecode, it must transfer CPU control to the native code that implements the bytecode. Because we want the virtual machine to act as much like a real machine as possible, we seek to minimize the overhead of this fetch and dispatch cycle. As a result, a significant number of very low-level (at the assembler or machine code level) programming tricks have been presented as options for control flow – something we normally take for granted. It turns out, of course, that for the very best performance, we must resort to machine code, and this means that the best technique varies from one processor architecture to another. In the next section, we briefly compare these subtle optimizations in the various dispatch strategies.

2.6 Virtual Machine Dispatch Strategies

The lowest-level and most fundamental job of the virtual machine is to dispatch instructions in sequence. There are several basic techniques to do this, and we enumerate them in the next section. They are all semantically equivalent; the differences between each are a matter of fine-tuning for performance or other criteria. Subsequently, we also discuss more advanced dispatch techniques.

2.6.1 Basic dispatch techniques

Function table In this technique, each bytecode has a corresponding C function that implements the bytecode's semantics. The address of each function can be kept in a table, indexed by the bytecode number. Alternatively, the bytecode program itself can be stored using the addresses themselves, instead of the bytecode index. That saves an array lookup. A C `for` loop fetches bytecodes, looks up the function, calls it, and repeats. Each bytecode is responsible for incrementing the virtual program counter, `pc`, by an appropriate amount, so that after the bytecode executes, `pc` is pointing to the next bytecode in the program. A less compact, but potentially faster form of the code does away with interpretation all together: instead of just storing addresses, the bytecode program is a series of native call instructions.

switch A large C `switch` statement has `case`s for each bytecode. Depending on how the C compiler generates code for the `switch`, this often turns out similar to the *Function table* approach, but uses a simple indirect jump through a table indexed by bytecode number, instead of a potentially more expensive procedure call. Here, again, a C `for` loop is required. After each `case`, control jumps over succeeding cases to the bottom of the `switch` and `for` loop, and then back to the head of the `for` loop. Some C optimizers may implement *branch threading* [27] to replace these two jumps with a single jump, from the end of each `case` to the top of the `for` loop. Alternatively, some virtual machines use C `goto` statements to hard-code this single jump at the end of each case.

indirect threaded Here again, an *indirect* jump instruction is used, but now it is placed at the bottom of each `case`, instead of at the top of a `switch` construct. There is a table mapping each bytecode to the native address at the start of the interpreter code that implements it. The bytecode is fetched from the program, and the address of its body is looked up in the table. Finally, we execute a jump to that address. This may be implemented with assembly language (two loads and an indirect branch), or using C language extensions that permit computed `goto`, such as those in `gcc`: `goto table[*pc]`. In other words, instead of jumping to the top of the loop for dispatch, dispatch is the responsibility of each instruction. Note, this makes each instruction slightly larger, but saves a branch.

direct threaded This is similar to *indirect threaded*, combined with the table-bypass refinement mentioned above in *function table*. The bytecode program is stored using native addresses for bytecodes, instead of the bytecode numbers. This saves a table lookup. The C computed goto code is simple: `goto *pc`. This amounts to a load and an indirect branch. Note that, since machine addresses typically are at least four bytes long, this makes the bytecode program less compact than if it's stored with only one byte values for opcodes. Furthermore, if virtual instruction operands are also stored as machine words, instead of one- or two-byte values, then all `pc` arithmetic and dereferencing will be aligned on word boundaries. This can improve performance on modern architectures.

If the bytecode program is stored on disk, a translation step may be necessary to rewrite the program instructions to use the machine addresses instead of the bytecodes, because the addresses are not available until after the interpreter is loaded. This could even be necessary if the bytecodes are stored in memory, if the front-end that generated the bytecodes uses single byte instruction codes. This translation is the approach taken in the virtual machine in GNU's `gcj` Java compiler system [3], which includes an interpreter in its runtime, in case classes are loaded from disk.

2.6.2 Advanced Dispatch Strategies and Other Optimizations

In addition to the simple variations on virtual instruction dispatch, several other more involved techniques have been developed to improve virtual machine performance. Next, we outline some of these. The first is not a dispatch technique at all, but rather a different kind of optimization.

Weak Operand Specialization Consider a virtual machine instruction that pushes a single small integer operand onto the stack, `push_int`. This is likely to be a very common instruction, and furthermore, certain operand values will be more common than others. We can augment the virtual instruction set with new instructions that "hardcode" these operands, — the new instructions take zero operands, but push a constant whose value is implicit in the instruction. For example, `push_0`. We call this *Operand Specialization*, or simply *specialization*, for short.

Smalltalk 80's virtual machine [11], for example, takes this approach. It offers 32 instructions, each with zero operands, to push any constant from -16 to 15 onto the stack. Among others, it also has eight 0-operand instructions to load the given operand object into one of eight local variable slots in a method. In the rare case there are more than eight slots, a 1-operand version is used that also takes the slot number as an operand.

We refer to this technique as *weak* operand specialization because it only specializes certain operand values, and only certain opcodes. In Section 3.3, we present a similar, but much stronger technique, which we simply call *Operand Specialization*, which does not have these limitations.

Super Instructions A virtual machine *superinstruction* is a macro instruction – a combination of two or more normal virtual instructions into a single new instruction. The semantic effect of the superinstruction is equivalent to executing the component instructions in sequence. For example, the two instructions `push_constant` and `add` could be combined into a single `add_constant` superinstruction.

There are at least three advantages to this technique. First, the virtual machine has a finite dispatch overhead. Suppose this overhead is 10 machine cycles of execution time. If, say, the `push_constant` and `add` instructions require 30 and 20 machine cycles to execute, respectively, then the cost of executing them is $10 + 10$ (two dispatches) $+30 + 20$, for a total of 70 machine cycles. With a superinstruction, only one dispatch is required, yielding a total of 60 cycles.

Second, the `add_constant` may take less than $30 + 20 = 50$ cycles to execute. Removing the dispatch branch leaves *straighter* code, and the processor may be able to execute it more quickly. Moreover, a peephole optimizer may be able to improve the code in the superinstruction, because the flow graph will be larger and less branchy. In this case, for example, four stack operations (`push1 1` requires a push, and the `add` must pop, read the value at top of stack, then write the sum), can be reduced to two (read value at top of stack, then write the sum.)

Third and finally, a bytecode program with superinstructions is slightly smaller to store, because the superinstructions takes up less space than their separate component instructions.

The vmgen [10] system facilitates the static creation of superinstructions as a part of its VM-building tools. This includes automatic inclusion profiling instrumentation in VMs, which can manually inform the designers' superinstruction selection.

Superinstructions allow us to group multiple bytecode instructions together. This leaves the problem of the operands of the component instructions. Typically, these operands are managed by catenating the operand bytes of each component instruction into a larger single array of bytes. This array becomes the operand of the superinstruction in the new bytecode program. The implementation of this superinstruction knows where to fetch the component arguments from within the array of bytes.

How do we choose which superinstructions to offer? The decision may be made statically, as part of the design of the virtual machine. Smalltalk does this, for example. Or perhaps the decision can be made still statically, at the time the virtual machine interpreter is constructed, but based on some dynamic profile of the one or more programs the interpreter is expected to execute. But clearly, if we choose these optimizations statically, we can only make a finite number of them, after which we will run out of bytecode representations.

This potential for combinatorial explosion is particularly evident if we think about combining these techniques. If we specialize 16 `push_constant` instructions, and combine these as superinstructions with 32 `store_local_slot`, we would have $16 * 32 = 512$ instructions for storing a constant into a local slot. And this example only accounts for superinstructions of length two!

Selective Inlining This scheme is proposed by Piumarta [33], as a refinement to direct threading, and exploiting the concept of superinstructions. The idea is to dynamically build the superinstructions at execution time. In this way, only the superinstructions (called *macro opcodes*) needed by the running program are constructed. Piumarta builds maximallysized superinstructions, but their lengths are severely constrained because of the design and implementation of his system, which we discuss in the next paragraph. Because they are created at runtime by simply copying machine code, no optimization of macro opcodes is possible, unlike with some superinstruction systems. Nonetheless, he reports a speedup of 1 – 2.0 for Objective Caml on the Sparc, running micro-benchmarks.

Piumarta's design means that macro ops must not span across basic block (single entry point, single exit point) regions of the virtual program. It is not possible to jump into the middle of a macro op, nor out of the middle. Therefore, all branches or branch targets force the end of a forming macro op.

Piumarta implements his macro ops by copying the native code which implements the virtual ops. While `gcc`'s labels-as-values extension has been available for some time, and intended specifically for interpreter implementation, his paper was the first to show that native code could be copied "out-of-line" and executed elsewhere. This inspired our technique, which is more general, as we will discuss in Chapter 3. Unlike us, he does not do any processing on the native code. Therefore, any code which contains native PC-relative addressing cannot be moved, and thus cannot be a component of a macro op. On many architectures, e.g. Sparc, this precludes ops containing a C function call. For low-level bytecodes this is only a moderate disadvantage, since these bytecodes' semantics are simple enough not to require function calls. For high-level bytecodes such as those used by Tcl, or low-level but rich object-oriented bytecodes, as in Java, this is a serious deficiency. We discuss Piumarta's work further in Section 6.5.

JITs Perhaps the state of the art software technique for virtual machine implementation is the *just-in-time compiler*, or JIT. Rather than just being an evolutionary refinement of highly optimized implementations for a virtual machine interpreter, the JIT focuses on actually translating bytecodes into native code. This is done at run-time, and indeed, compilation is usually deferred until execution of individual methods, or even smaller units of code. An adaptive JIT may apply varying levels of optimization during initial or subsequent compilation, sometimes guided by dynamic profiling collected by dynamically inserted instrumentation. At the extreme bottom end of optimization, some JITs also interpret some code some of the time — the code may not be executed frequently enough to warrant time spent in compilation. Interpretation also makes profiling easier, since native code does not need to be instrumented.

As expressed earlier, JIT translation may be too time-consuming for some applications, especially interactive use of scripting tools such as Tcl. Our system attempts to reap some of the benefits of native code, without requiring a lengthy compilation step. It is a new design point in the range between interpretation and optimizing compilation. In addition to being useful as the only execution engine in a virtual machine, it could also be added to an adaptive JIT, since it could might execute faster than the interpreter, but compile faster than even the zero-optimization level compiler.

2.7 Native versus virtual machine performance

Compared to native execution, interpretation faces two fundamental performance problems. Because the interpreter is a *virtual* machine, it must expend software execution cycles on tasks that hardware machines get "for free", at least in terms of cycles required. These are (i) a dispatch loop to fetch the virtual instructions and sequence their execution, and (ii) instruction decode logic for each virtual instruction, to determine its format, fetch its operands, etc.

In Section 2.6 on virtual machine dispatch strategies, we described some of the techniques used in interpretation. They all consume some significant proportion of machine cycles on these dispatch and decode tasks. In a native machine, substantial hardware resources are used to implement fetch, decode, and dispatch logic largely in parallel with execution of the actual work of instructions - e.g. fetch can happen concurrently with an ALU operation.

In a virtual machine, instead, real cycles must be expended before any work can begin. Furthermore, modern native machines have complex features such as pre-fetch, branch prediction, instruction cache, and out-of-order superscalar execution to run native code workloads. Interpreter workloads have a different character than typical native workloads, and thus cannot fully exploit the benefits of those features. Worse, they may even run afoul of the features. [9].

In addition to this fundamental fetch/decode difference between native and virtual machines, a number of micro-architectural features are implicated in the interpreter design choices enumerated in Section 2.6. We briefly summarize these now.

2.8 Micro-architectural implications of virtual machine designs

A virtual machine interpreter is an unusual type of workload to present to the processor hardware. Various features of certain dispatch techniques — e.g. use of calls versus indirect branches — interact differently with architectural features of modern processors. Ertl [8] presents an informal but empirical discussion of the performance comparison of these techniques. Now, we briefly consider the impact of virtual machine design on each of several classical processor architectural techniques.

2.8.1 Branch prediction

Even in straight-line bytecode, the processor executes at least one native branch per virtual instruction, as a part of the virtual dispatch loop. Thus, branch predictor and branch target buffer design are very significant. Because the virtual machine workload is atypical, some processor designs do poorly [9]. Dispatch typically uses some form of native indirect branch, which is challenging for branch prediction hardware. Note that the branch target buffer can do a better job on direct-threaded code than switch dispatch. With direct threading, each virtual instruction has at least one branch dispatching to the next, whereas with switch, dispatch for every VM instruction is concentrated through a single native indirect branch. The branch predictor uses the native instruction address as the most significant part of the key it uses to predict a target address, and thus it will be wrong most of the time [9].

2.8.2 Instruction-level parallelism

Recall that one challenge to good performance in virtual machines is maintaining interpreter meta-state, such as virtual program counter and stack pointer, in software. One useful technique is to schedule the native instructions responsible for updating this state so they are interspersed with the semantic work of a virtual opcode. As data dependence allows, this can increase instruction level parallelism (ILP) and reduce the latency introduced by dispatching to the next virtual instruction.

2.8.3 Cache

Compared to native execution of code compiled from high level languages, interpreters have an interesting impact on caches in the interface between CPU and memory system. Caches are necessary because CPU clock rates and computation bandwidth are much higher than main memory clocks and bandwidth. Most modern CPUs use a so-called *Harvard* architecture [13], with two separate level one (closest to the CPU) caches for instructions and data. These can each be interfaced efficiently to different parts of the CPU, and instruction cache (I-cache) semantics are slightly more relaxed, with different implementation trade-offs than data- (D-) caches, because instruction memory transactions are overwhelmingly read-only.

Interpreters in general conflate the roles of the I- and D- caches, because the instructions being interpreted reside in D-cache. The I-cache only serves the interpreter itself, and its runtime system and libraries. Thus, the load on the I-cache is slightly less than normal, and the D-cache does double duty. Worse, D-cache logic for read-write consistency is wasted when relatively static bytecode instructions are stored there.

Our interpreter technique, on the other hand, returns demand to the I-cache, since we translate bytecode to native code, and execute that. However, we not only return demand to the I-cache, but increase it beyond normal levels. This means I-cache is a critical factor in the performance of our system. We discuss this more in Chapter 5.

Our technique for virtual machine implementation, *catenation* with *operand specialization*, lies somewhere between Selective Inlining and JITs. We describe its design in the next chapter.

Chapter 3

Catenation and Specialization

Our goal is to decrease execution time of Tcl scripts by compiling the Tcl bytecodes to native code. This goal is subject to a number of constraints. First, we want to perform the compilation at run-time so we don't compromise the interactive modality of Tcl usage. Second, we want to leverage existing and future work on the current interpreter, which correctly implements the semantics of each bytecode. Finally, we want to accomplish this with a minimum of programmer effort, which consumes time and introduces the possibility of incorrect semantics.

A full-blown JIT compiler requires too much programmer effort, does not reuse adequate amounts of interpreter code, and introduces substantial compile-time (and thus run-time.) delays for small scripts and interactive use. Instead, our approach is a new design-point on the spectrum between interpretation and compilation. We preserve as much code from the virtual machine as possible, including the entire run-time system and library, and even most of the interpreter loop, by crafting a simple compiler directly from the existing interpreter. In this Chapter, we present two techniques to undertake this compilation.

3.1 Interpreter Performance

Section 2.7 showed how virtual machines are fundamentally different from native machines because they perform instruction fetch and decode in software. This, then, is the first problem our system addresses to improve interpreter performance via compilation.

```
goto *ip++;
```

(a) C code for minimal dispatch strategy (direct threading).

```
load  r2 = [r1]
add   r1 = r1 + 4        ; assuming machine uses 4 byte addressing
jump  r2
```

(b) Corresponding native code. Virtual instruction pointer `ip` is allocated to register `r1`.

Figure 3.1: Overhead in native instructions for minimal dispatch strategy.

The particular burden that the instruction dispatch loop imposes can be seen by looking at the assembly code (Figure 3.1a) for the most efficient possible dispatch, direct threaded code (Figure 3.1b.) The interpreter must, at a minimum, load the address of the next opcode to be executed, jump to it, and increment the virtual instruction pointer. The overhead in most interpreters is often considerably higher, but even these minimal operations — a load and two branches (one direct and one indirect) — are extra work compared to native execution.

The simplest possible compiler must, at least, eliminate this extra work. Our strategy, which we called *catenation*, addresses this, as we describe below.

The further burden that instruction decode imposes is exemplified in Figure 3.3. This figure shows an `int_add_immed` opcode for a hypothetical virtual machine, and its implementation in an interpreter. This opcode adds an immediate integer operand to the integer stored in a virtual accumulator. Because the operand is an *immediate*, it is, by definition, a part of the virtual instruction. A native machine fetches the immediate operand of a native instruction as a part of the instruction; no additional cycles are necessary. But in a virtual machine, the instruction body itself, after it has already begun executing, must issue and wait for a memory load before it can begin the work of the addition.

Clearly it would be beneficial if such immediate operands were truly immediate, that is, implicit in the native instructions used in the bodies of bytecodes with virtual operands. Our *operand specialization* strategy makes virtual immediate operands into native operands, avoiding the extra decode cost.

3.2 Catenation

The bulk of the interpreter loop is contained in giant `switch` statement, with cases for each virtual opcode. We call the code in these cases *instruction bodies*, or just bodies, for short. The job of the dispatch loop is to sequence the execution of these bodies according to the instructions in the bytecode program. That is, the dispatch loop executes one body after another. The overhead of dispatch is any work other than that performed in the bodies themselves.

Let us illustrate this with a simple example. Figure 3.2a shows a short bytecode program fragment, composed of three bytecode instructions. Note that the `push` opcode is used twice. The native code bodies for each of these instructions is shown in Figure 3.2b. We also show the native code executed for each instruction dispatch by the interpreter loop. In Figure 3.2c, we show the dynamic sequence of native instructions executed when interpreting this bytecode program.

Notice, in this example, that a significant portion of the native instructions are for dispatch. These are overhead instructions. The instructions shown in bold in Figure 3.2c are the only native instructions that contribute to useful work. Now we are set to understand the idea of "catenation." If we simply copy into executable memory the sequence of useful work instructions — those shown in bold — we have "compiled" a native code program with exactly the same semantics as the interpreted version. This program will have eliminated all dispatch loop overhead.

Of course, most programs contain branches and loops, so dynamic execution paths do not look, as they do in this case, exactly like static code in memory. Catenation handles control flow by changing virtual machine jumps into native code jumps. Each virtual instruction is compiled into a series of native instructions at a well-defined address range in memory, and thus the native jump targets the start of this range. Any interpreter event where execution must take a non-linear path, including exception handling, is handled similarly, as we explain in Chapter 4.

```
push  2
push  3
add
```

(a) Sample bytecode program. Note, opcode `push` is used twice, with different operands.

```
push:
    p1
    p2
    p3

add:
    a1
    a2
    a3
    a4
    a5

dispatch:
    o1        ; jump to top of interpreter loop
    o2        ; ld [ip] to get opcode
    o3        ; increment ip
    o4        ; jump indirect to instruction body
```

(b) Definitions of virtual instruction bodies in native code. Each p_1, a_2, etc. represents a native instruction.

$o_1 o_2 o_3 o_4$ $\boldsymbol{p_1 p_2 p_3}$ $o_1 o_2 o_3 o_4$ $\boldsymbol{p_1 p_2 p_3}$ $o_1 o_2 o_3 o_4$ $\boldsymbol{a_1 a_2 a_3 a_4 a_5}$

(c) Dynamic sequence of native instructions executed when interpreting program in (a).

$p_1 p_2 p_3$ $p_1 p_2 p_3$ $a_1 a_2 a_3 a_4 a_5$

(d) Static sequence of native instructions emitted by catenating compiler for program in (a).

Figure 3.2: *Compiling* bytecode objects into catenated copies of native code from interpreter avoids dispatch overhead

3.2.1 Benefits of Catenation

Catenation itself does not synthesize or even transform much native code. It just makes copies of the native code produced by the C compiler when it compiled the interpreter. When we discuss problems with this approach, below, we point out that this necessarily means the compiled code itself will not be much faster than interpreted code. It just reduces dispatch overhead, as described. However, it does have two key advantages over a more sophisticated compiler: it is faster and simpler.

The catenating compiler can run very fast because most of its work consists of nothing more than copying native code bytes into memory. Compilation speed is very important for any just-in-time compiler, because the compile time is a part of execution time. It is particularly important for an interactive scripting language, since long delays will be perceptible to the user, who expects instant response to commands.

The catenating compiler is simple because, ideally, the compiled instruction bodies will be very nearly identical to those in the original interpreter `switch` statement. Thus, a minimum of extra effort is required to emit native code. Furthermore, semantics are preserved, and the entire existing virtual machine run-time and library system is preserved and re-used. Future changes to the interpreter can be easily leveraged in the compiler.

As we shall see in Chapter 4, where we discuss our implementation, most of the changes required for catenation, *per se*, are confined to the interpreter dispatch code. Thus catenation itself does not require modifications to the implementations of all of the approximately 100 bytecode instructions.

In addition to these two main benefits of simplicity and high speed compilation, catenation yields several other useful consequences. The first of these is to enable our second key technique for enhancing interpreter performance, operand specialization. While the interpreter has only one generic implementation of, say, the `push` opcode, an catenated program has a separate version for each static instance of that opcode in the program. This means we can customize each version to improve performance. We describe operand specialization in Section 3.3.

Another useful consequence of catenation is a much straighter dynamic execution path, because we eliminate dispatch and lay native instructions out in memory for each static virtual instruction. By *straighter*, we mean it contains less conditional branches. This means that the processor's branch predictions may improve. The processor's out-of-order execution also has longer straight-line blocks of code to work with, and thus may also perform better.

In addition to removing the expensive indirect branches in dispatch (o_4 in Figure 3.2b), we also remove a memory load (o_2.) Loads are real work for a processor and memory system, and their removal improves performance. Better still, the indirect branch is dependent on the results of the load, presenting a real barrier for superscalar execution.

In summary then, catenation is a simple idea with relatively simple implementation. In can run very quickly. It removes overhead instructions from the interpreter dispatch loop. The instructions it removes are particularly difficult obstacles for efficient execution on a modern microprocessor, and the resulting code is well-suited to further performance improvements. Of course, most of these benefits come with some converse costs. We discuss these in the next section.

3.2.2 Challenges to Catenation

Our goals for the catenation strategy fall into two categories. The first is, of course, making the code go faster. The second is to craft a simple, efficient system with minimal effort. We will address the problems in each of these categories in turn.

As sketched above catenation does reduce the number of dynamic instructions executed by bytecode programs. However, it has the potential to degrade performance, too. First, as with all just-in-time compilers, compilation time is part of our execution time, and if execution time is too short, the compilation effort cannot be profitably amortized over execution. Catenation has this effect, but is so efficient that even relatively short Tcl programs can benefit.

Second, and more importantly, the obvious effect that catenation has on the bytecode program is a significant static code size expansion. Bytecode is designed specifically to be very compact — typically more compact even than native code [44]. Catenated code has exactly the opposite character. This is because potentially many copies are made of each bytecode instruction body, as dictated by the length of the bytecode program. In Tcl, in particular, the bytecode bodies are already very high-level and large, so catenation can make the emitted native code very large.

The resulting burden on I-cache throughput and locality can be significant. The performance tension between poorer I-cache performance but fewer actual instructions executed is the major focus of our experimental evaluation in Chapter 5. In a smaller but related effect, the larger footprint in instruction memory can significantly exercise bottlenecks in the virtual memory system, including the TLB (translation lookaside buffer) and actual traffic of paging in from disk. However, this effect is both coincident with, and much smaller than, the I-cache problem, and so we focus on that.

We can mitigate some of the problems of catenation by extracting even more benefits from the catenated code. As mentioned above, such code has special properties that make it amenable to low-level optimization. We exploit distinct copies of each bytecode in catenated code to apply custom specialization to each copy. This is described in detail in the next section.

3.3 Specialization of operands

In compilers literature, *specialized* code is code that is less generic [15]. That is, its semantics are determined more by the code itself, and on a more local level, rather than depending on data, or distant code. Specialized code can execute faster, because it does not have to fetch the data needed to drive its semantics. Because values and control flow, for example, are more deterministic, with each instruction less dependent on loads or earlier instructions, the code can be executed faster, and with high instruction-level parallelism.

Our system mainly exploits only one form of specialization. We substitute bytecode operands into the catenated instruction bodies as native immediate operands in the native code. These operands are, in the parlance of traditional optimization, *run-time constants* [12]. That is, the operands are variables in the generic interpreter code, but, because we generate custom native code, for *each* bytecode instance, late in the compilation/execution process, we know the constant values of many of the operands. We can then emit code that contains the constants instead of references to variables.

For example, consider the instruction body for an `int_add_constant` opcode, shown in

```
ld    [pc + 1], r1       ; this operand-load will be specialized away
add   accum = accum + r1
```

(a) instruction body of imaginary opcode `int_add_constant`

```
add   accum = accum + 2
```

(b) instruction body for specialized opcode `int_add_constant 2`

Figure 3.3: Specialization replaces memory loads of generic operands with immediate constants in native instructions

Figure 3.3a. Let us suppose it takes one operand, an integer constant, and increments an accumulator by the constant.

Even though it takes a constant value, the value is not implied in the opcode. The instruction specifies it as an operand. Multiple instances of this instruction in a bytecode program, e.g. "`int_add_constant 2`" and "`int_add_constant 3`" may have different values for the operand. Thus, the instruction body must be *generic.* The generic version loads the actual constant from the instruction in the bytecode stream. However, after catenation, a *separate copy* exists for each static instance of the instruction. These copies can be *specialized* to use this run-time constant, and avoid the memory load, as shown in Figure 3.3b.

There are several ways for instructions to acquire operands. Some opcodes take no operands, or the operand may be somehow implied by the opcode itself, or taken directly from an accumulator or an execution stack. For other opcodes, operands can be of many different types. Our example shows the simplest type, a literal immediate integer, used, for example, in an arithmetic instruction. These integers may be signed or unsigned. Other operands may be immediate integers, but not really used as literals. Instead, they may index a table, e.g. the *literal table* in Java or Tcl, where the actual object (usually stored as a native machine pointer) or value is stored. Using the value as an operand requires dereferencing this single layer of indirection. Integers may also represent counts, such as the number of operands to pop from the stack for a computation, or function call.

Our operand specialization technique must handle all these types, and understand the sizes, signedness, and some semantics, of each. This is necessary to correctly emit or fill in specialized instruction bodies with values that implement the correct semantics of the original, generic opcode.

Typical specialization techniques work on an intermediate representation of the code (IR). Our compiler does not use an IR. Instead, we catenated the native instruction bodies at a very low-level, into new executable memory. To implement operand specialization, we consider the instruction bodies somewhat less opaquely than catenation. Instead of unchangeable sequences of code, we treat the bodies as *templates*. They are of fixed length (measured in instructions and bytes). The only fields of the instructions we allow to vary are the literal immediates. Thus, we do not change register numbers, for example. This is discussed in more detail in Chapter 4.

Other transformations in our system also take on the character of specialization. For example, our conversion of the indirect jumps, used in the instruction bodies of the virtual `jump` bytecode, to direct native branches is a kind of specialization. Furthermore, the *unswitching* we perform when eliding the dispatch loop — can be viewed as specialization on the variable *opcode*, which is a known constant at each specific point in the catenated program. However, we treated unswitching and native jump conversion as parts of our catenation strategy.

3.3.1 Advantages of Operand Specialization

As with catenation, fixed-length instruction bodies mean that operand specialization can run at high speed. Changes may be necessary to each template, as we read operands from the original bytecode stream, and substitute them into the instruction bodies. The number of substitutions is a linear function of the number of instructions in the bytecode program we are compiling. Each substitution typically involves only a few branches, loads, shifts, and stores. This may be an order of magnitude more work than the simple copying (load, store, increment) performed by catenation, but is also much faster than real transformations in a more sophisticated compiler using an intermediate code, and subsequent translation from intermediate to native code. On the other hand, the fixed-size constraint on each instruction body means we can avoid complex multi-pass linking steps, and memory allocation of output buffers is very efficient.

As with catenation, we want to minimize the effort required to change the interpreter into a specializing compiler. Our simple specialization templates make this possible. However, it is still a significant amount of programmer work, as we discuss below.

Operand specialization makes the total number of instructions *smaller*, which also saves space and time. Of course, the catenation that lays the groundwork for specialization makes code much larger, but at least the operands themselves consume less room, as they are packed tightly into native instructions.

In addition to packing the operands into instructions, we are also able to eliminate other instructions, such as those that load the generic operands from the bytecode stream. For example, if the operands are from a table of literals, we can do this table lookup at specialization time, saving more time and space avoiding the dereference. This lookup optimization is also applied to bytecodes with secondary opcode selectors, like `math_func`, which takes an integer argument to indicate which math operation to perform, for example `sin`, `sqrt`, etc.

Thus, operand specialization reduces static instructions (makes the code smaller), dynamic instructions (fewer instructions execute at run time), and also the number of memory loads (since loads from memory are replaced by loads of immediate constants.) All of these things favor faster execution.

3.3.2 Challenges to Operand Specialization

Operand specialization is more difficult to implement than catenation. The instruction body for each bytecode which takes operands must be modified to become a suitable template. There are many such instructions, some with more than one operand, and, as discussed above, many types of operands.

Our approach to generating these templates is discussed in more detail in Chapter 4. Briefly, as with catenation, we use the C compiler, and the C code for the generic bytecode interpreter, as a basis for building templates. In Chapter 4, we shall see that this technique entails considerable design and debugging effort to work around the correct, but unsuitable, semantics of the C compiler for this purpose. Still, it is important to note that these difficulties are not weaknesses of operand specialization, *per se*, nor even our approach of fixed-length native code templates.

Rather, they are a challenge presented by our particular choice of implementation tools for creating the templates.

While the run-time spent on catenation is quite low, involving copying data in a fixed-size template, specialization requires considerably more work, interpreting and executing each change required in the template, for each operand in the bytecode program. Thus, the compile-time spent on specialization is expected to be larger than for catenation.

Finally, specialization would not be possible without catenation; normally interpreted code cannot be specialized because the instruction bodies are generic. As discussed above, catenation dramatically increases static code size, which can cause slower execution. So, while specialization itself improves executing code, its requirement for catenated code *may* result in overall slower code.

Before we conclude our discussion of operand specialization, we should note that many other kinds of specialization are possible, and might be facilitated by availability of catenated code. These include using *type inference* [25] to specialize on the type of an object, and thus elide virtual dispatch code. This can yield further optimization opportunities, as the control flow graph is straightened.

Perhaps the most important implication of operand specialization, in our system, is that it allows us to compile away the bytecode. In this sense, we are a true compiler, because the executing code does not refer to the bytecode stream after it starts running.

3.4 Summary

Our system lies just on the boundary between interpretation and compilation. It is a true compiler, but does not undertake any abstract representation nor transformation of the code. Our goal was to make the Tcl virtual machine interpreter run faster, and, in that sense, it is perhaps better to consider the system a highly evolved interpreter.

We preserve the existing run-time of the interpreter, and, in particular, base our work on the implementation of the existing interpreter. Our two key techniques to improve interpretation are catenation, which eliminates instruction dispatch overhead, and operand specialization, which shortens the code, and saves memory loads of operands. Specialization depends on catenation to first copy generic instruction bodies into individual templates for each virtual instruction in a bytecode program.

These simple techniques can compile very quickly, preserving the interactive feel of a scripting language, and allowing for rapid amortization of JIT compiling costs over even relatively short execution times. Compared to a more sophisticated compiler using an intermediate code and optimizations, the generated code quality will be inferior. However, the compile time, and the engineering effort in constructing the compiler itself, is improved. By using the interpreter as the basis for the compiler, correct semantics for existing and future bytecode instructions are easily preserved.

In the next chapter, we will present the details of our implementation of these two key ideas, catenation and specialization.

Chapter 4

Implementation

In this chapter, we detail the implementation of our design into an efficient working system. The result is a drop-in replacement for the stock Tcl interpreter, appropriate for running all Tcl scripts. Its semantics are identical to the stock interpreter, differing only in performance, with no noticeable compromise in start-up time or interactive responsiveness.

We'll first present the implementation of the core of the just-in-time compiler itself. The compiler rapidly translates the bytecode program using a native code "template" for each virtual instruction. Our presentation of the compiler proper largely assumes these templates already exist, but describes the data structures in which they are defined and stored, and then how they are used to compile.

In the latter sections of this chapter, we show how the templates are synthesized. Aside from the compiler itself, the other key part of our contribution is our engineering process in crafting the templates from the existing virtual machine interpreter and the C compiler. We believe this approach is efficient, and it introduces fewer semantic errors than other compilers which re-implement, rather than reuse, the interpreter.

Our system is implemented on the Sun Sparc microprocessor, running the Solaris operating system. We target the 32-bit `sparcv8` instruction set [14]. We use Tcl version 8.4 as the basis for our JIT compiler, and also as the baseline for performance evaluation.

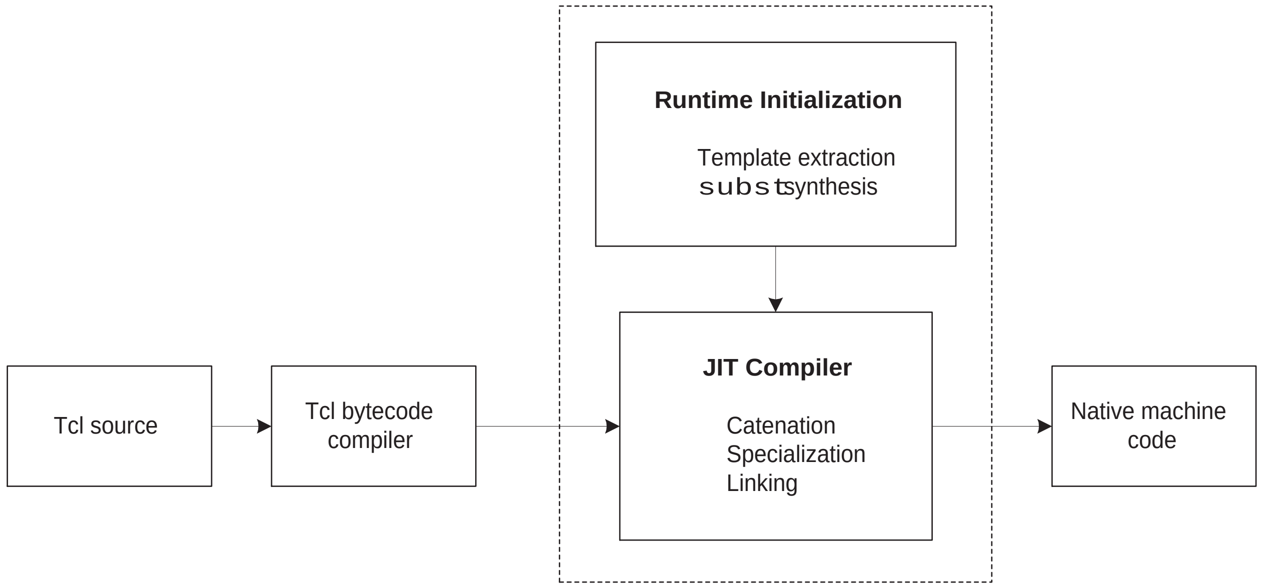


Figure 4.1: Compilation stages in Tcl JIT

4.1 Core JIT Compiler

Refer to Figure 4.1 for a conceptual diagram of the compiler, and its place in the Tcl system. The compiler realizes speed and simplicity using the ideas of catenation and specialization. The working system requires infrastructure to support implementation of these ideas. The speed and simplicity of the system derives from fixed-length sequences of native code called *templates*, and a set of pattern and actions called *substs* which facilitate quick manipulation of the templates. To further increase speed, these structures are initialized once at interpreter start-up time. The JIT compiler can then rapidly translate bytecode to native code at execution time.

In this section, we define what templates and `subst`s are, and describe the data structures we use to store them. We'll then show how these structures are populated, and finally, but most importantly, how the templates are actually used to realize catenation and specialization.

4.1.1 Templates

The template is a sequence of native instructions that implement the semantics of the opcode by manipulating the state of the Tcl interpreter and calling into its run-time library. We require that each template has various properties. The template code must be contiguous, because we extract it from beginning to end. Intervening code would cause severe code bloat during catenation, and our analysis of the template during specialization would have to deal with irrelevant pattern matches.

To increase the performance of compilation, we pre-compute as much information as possible about the templates, so that catenation and specialization can proceed with minimum effort. The templates and the pre-computed information are stored together in data structures which we describe now.

4.1.2 The `subst` data structure

Along with the template, we keep a list of actions that must be taken to specialize it for the appropriate operands in the bytecode instruction, and other transformations that need to be made to the template to fit it into the resulting native code output. All this data is stored in a C data structure `instruction_desc_aux_t`, shown in Figure 4.2 together with related structures. We now describe these in detail, since they lie at the center of our implementation.

In addition to the native code template, the `instruction_desc_aux_t` also stores meta-information about the virtual opcode. The information includes the length in bytes (structure field `bc_bytes`), including operands, of a given opcode, when in a bytecode program. For example, an `incr_scalar1` opcode takes two bytes, one for the opcode itself, and one for the single-byte immediate operand. We also store the number and types of operands (fields `bc_operands` and `op_types`.) The size of each operand, which we also need to know, is implicit in its type. Together, this information allows us to derive the offset in bytes of the start of each operand from the start of the bytecode instruction.

The template's native code is described simply in two fields: native address (`ntv_start`) and length (`ntv_len_src`.) The address is a pointer into the executable memory of the running Tcl interpreter process. The raw code for the templates exists in this process, as described in Section 4.2.

While `ntv_len_src` tells us the number of bytes we must copy *from* the template into a catenated code unit, we also store a separate `ntv_len_dst` field, which is the number of bytes to *allocate* to this opcode body during catenation. This may be slightly larger than the

```
typedef uint tcl_opcode_t;
typedef uint *iaddr;

typedef enum {
  SIT_ARG, SIT_LITERAL, SIT_BUILTINT_FUNC, SIT_JUMP, SIT_PC, SIT_NONE
} subst_in_type_t;

typedef enum {
  SOT_SIMM13, SOT_SETHI, SOT_JUMP, SOT_CALL
} subst_out_type_t;

typedef struct {
  subst_in_type_t subst_in_type;   /* how to extract operand from bytecode insn */
  subst_out_type_t subst_out_type; /* bit ops necessary to specialize or fixup native template */
  uint ntv_offset;                 /* where in template to make substitution */
  uint bc_offset;                  /* where in bytecode insn to grab operand data */
  uint bc_size;                    /* size of bytecode operand data */
  bool_t bc_signed;                /* flag: is bytecode operand data signed? */
  uint ntv_offset2;                /* where in template to make optional 2nd substitution */
  uint jump_insn;                  /* native branch insn to use, for subst_in_type == SIT_JUMP */
  int mul;                         /* multiply bytecode operand by this before subst */
  int add;                         /* add this to bytecode operand before subst */
} inst_subst_t;

typedef struct {
  char *name;
  tcl_opcode_t opcode;
  uint bc_bytes;
  uint bc_operands;
  InstOperandType op_types [MAX_INSTRUCTION_OPERANDS];
  iaddr ntv_start;
  uint ntv_len_src;                /* length of template in bytes */
  uint ntv_len_dst;                /* length, including extra insns we append */
  subst_list_t subst_list;         /* list of substs for this template */
} instruction_desc_aux_t;
```

Figure 4.2: Data structures for `substs`, which direct catenation and specialization of each opcode

template, because we append synthesized native instructions to bodies for certain opcodes. We also exploit this facility during development, to introduce extra instructions for profiling or debugging instrumentation.

The three `ntv_` fields are used for catenation proper, that is, copying templates into new executable memory. All the other fields are for specialization.

Finally, we store pointers into an array of `subst`s. These form, for each opcode, one list of zero or more `subst`s. Together with the fields describing input operands, these allow us to implement operand specialization. Referring to the `inst_subst_t` struct, we see the heart of the specialization system. Each `subst` makes a small change in the catenated code, after it is copied into place. The type of change is controlled by the `subst_out_type` field; the various types are enumerated in Figure 4.3.

Each one of these `subst`s corresponds to a given location in the template. This location is indicated by the `ntv_offset` field of the `subst`. Except for `SOT_CALL`, each of the output types require additional parameters to fully evaluate the `subst`. `SOT_SIMM13` and `SOT_SETHI` require immediate integer values. In each case, the arguments come from the bytecode stream. The field `subst_in_type` defines the exact size and type of argument to extract from the bytecode, and potentially some transformations on the value after it is extracted, but before it is passed to the `SOT_` handlers. The possible values for this field are shown in Figure 4.4.

In the case of `SOT_JUMP`, there are two arguments. The first is `jump_insn`, which indicates which Sparc branch instruction to use. The three choices are `sparc_branch_always`, `sparc_branch_on_zero`, and `sparc_branch_non_zero`, which correspond to the actual Sparc instructions `b`, `beq`, and `bne`. The appropriate value is filled in at `subst` creation time, using table-driven (i.e. hardcoded) logic based on virtual opcode.

The second argument for `SOT_JUMP` is the branch target, which is computed from the bytecode input, which will have type `SIT_JUMP`. Both Tcl VM jumps and native Sparc branches are pc-relative. So, the VM `pc` of the current bytecode is added to the given jump offset to compute the actual bytecode destination. This is translated through a linker "offset" map (see Section 4.1.4.1, below) mapping from VM `pc` to native `pc`. Finally, we subtract the current native address (i.e., the address of the branch instruction we are emitting) from this target

`SOT_SIMM13`	Place a 13 bit immediate signed integer constant into the low-order bits of a Sparc instruction
`SOT_SETHI/OR`	Set a 32 bit immediate integer constant into a `sethi/or` instruction pair
`SOT_JUMP`	Write a branch instruction of a given type, with target directed at a given address
`SOT_CALL`	Fix up a `call` instruction after it was relocated during catenation, so that its target is the same as it was in the template

Figure 4.3: Output types for `substs`

`SIT_ARG`	a plain operand from the bytecode stream
`SIT_LITERAL`	an operand from the bytecode stream, translated through the literal table
`SIT_BUILTIN_FUNC`	an operand from the bytecode stream, translated through the builtin math function table
`SIT_JUMP`	the destination VM PC of a VM jump opcode
`SIT_PC`	the VM PC of the VM opcode currently being compiled
`SIT_NONE`	no input is required for this `subst`. Currently used only for `SOT_CALL`.

Figure 4.4: Input types for `substs`

native pc, to yield a pc-relative Sparc branch offset.

For `SOT_SIMM13` and `SOT_SETHI`, the argument again comes from bytecode operands, but is an immediate value, rather than a branch target. `SOT_SETHI` is unlike the other output types, in that it patches two different native instructions. The Sparc idiom for setting a 32-bit constant in a register requires first a `sethi` of the most significant 22 bits of the register, followed by an `or` of the 10 least significant bits of the same register. The `or` may be scheduled anytime after the `sethi`, but before any other definition or use of the register. During `subst` table generation, we scan forward from the `sethi` searching for the corresponding `or`, ensuring we match the same register, and before any intervening reference. The location of the `or` instruction is recorded in the `ntv_offset2` field, whereas the location of the `sethi` is stored in `ntv_offset`. Both offsets are relative to the start of the opcode body. All other `subst` output types use only `ntv_offset`, ignoring `ntv_offset2`.

There are one or more `substs` for each bytecode operand in a bytecode instruction — some instructions take more than one operand. The number of `subst`s per operand is usually exactly one, but in some cases, an operand needs to be specialized into more than one location in the opcode body, because the C optimizer employed rematerialization or constant propagation, or the C code was deliberately written to use the operand more than once without first placing it in a variable.

The `bc_offset` field expresses the offset of the first byte of the operand, in bytes from the start of the virtual instruction (i.e., from the current VM pc during translation.) `bc_size` indicates the number of bytes that need to be extracted from the bytecode program — that is, the size of the operand. Finally, `bc_signed` is a flag, set when an operand less than 32 bits in size requires sign extension.

The `mul` and `add` fields of a `subst` are rarely used. During specialization, after each bytecode operand is extracted, it is multiplied by the constant in `mul`, and then added to the constant in `add`. Usually these values are set to the respective arithmetic identities, i.e. 1 and 0. They are used to adjust for C compiler optimizations. For example, if the C code increments a pointer to an object (or, equivalently, indexes an array) by some value x, and the size of the the referenced object is four bytes, the actual increment will be $4x$. In this case, we must multiply the operand by four, before specializing into the template.

4.1.3 Extracting templates in running interpreter

The initialization pass builds the templates and `subst`s by matching patterns in the runtime (i.e. running!) executable memory image of the Tcl "interpreter". To find native code for a template, this pass consults the array `jump_range_table` (see Section 4.2.3), which contains the starting and ending memory addresses for the instruction body of each opcode in the interpreter.

The initialization phase uses this information to populate the template part of the `instruction_desc_aux_t`, and this enables catenation. To implement specialization, we must also fill in the `subst`s. After establishing the native address range of the opcodes, the initialization pass scans each, from start to end, for certain bit patterns that indicate specialization points. We call these patterns *magic numbers*, and they are simply distinctive integer constants. We place these values in the C source code, and ensure they are present at locations appropriate for operand specialization, even after passing through the complicated machinery of the software build process — the C optimizer, in particular. We elaborate this aspect of the implementation in Section 4.2.2.

The scanning requires work linear in the size of the native code, but we use several passes, and it does take some time. This is a key reason the scanning work is done only once, in the initialization phase, and cached for subsequent use by the JIT compiler. Otherwise, it would have to be performed each time we compiled a bytecode object. Indeed, an early implementation operated in this way. On long-running procedures, it still performed better than pure interpretation, but the present implementation allows profitable compilation of much shorter-running procedures, even if they are large and thus require relatively more compilation time.

The initialization pass of our JIT compiler runs when the Tcl core library is itself initialized. This is usually when a Tcl script is invoked as a new program by the operating system, but could occur at other infrequent times, such as when a program using an "embedded" virtual machine starts or re-initializes itself.

4.1.4 JIT Compilation

The stock Tcl virtual machine calls on the bytecode compiler to compile a Tcl source code object (a string) to bytecode the first time the source is ever executed. The result is cached "permanently". That is, it does not expire after a specific period of time, but, under certain circumstances, the cache is invalidated, and subsequent executions require that the code be re-compiled. This can happen, for instance, if assumptions about certain invariants the compiler made change. For example, if source was initially compiled in one Tcl namespace, and then used in another, it must be re-compiled.

After the bytecode compilation, and on all subsequent executions of the source code object, the normal virtual machine interprets the resulting bytecode. We modify this system so that the interpreter never interprets bytecode, but, instead, jumps to a native code translation thereof. This translation occurs immediately after the bytecode compilation, and at no other time. In this section, we describe the implementation of the native code translation that runs each time a new bytecode object is compiled.

Recall from Chapter 3 that our compiler is based on two key techniques, catenation and operand specialization. In Section 4.1.3 we showed how a single pass, running once at initialization time, builds the template and `subst` data structures which form the basis of catenation and specialization. The structures are, in essence, simple “instructions” about how to catenate and specialize the templates. The JIT compiler proper actually *executes* the “instructions”, via interpretation.

4.1.4.1 First pass

Given a bytecode object, the native code compiler makes two passes over the virtual instructions. Each pass looks at each virtual instruction in program order. The first pass determines the overall size of the eventual native code output, so that a block of executable memory can be allocated. The compiler emits native instructions into this memory. This pass also builds an “offset” map, recording the memory address at the starting point of the native code block corresponding to each virtual instruction. This map is consulted as a part of the “linking” phase in the second pass. Aside from the bytecode program itself, the key information required to build this map is the size of the native code extent of each virtual instruction. Recall that this is stored in the `ntv_len_dst` field of the template, as described in Section 4.1.3.

4.1.4.2 Second Pass

Once the first pass of the compiler computes the sizes and offsets of all the instruction bodies, and has allocated the executable memory into which we place them, we are finally set to actually emit code. We do this in our second pass, using our two key compilation strategies, catenation and operand specialization.

Catenation Once all the preparatory work is done in the initialization phase and first pass of the compiler, catenation amounts to simply copying data. Thus, it runs very quickly, and with little I-cache footprint. It copies native code bytes from the template corresponding to the virtual opcode, followed by “fix-ups” necessary to ensure the code works even after it is moved in executable memory.

Recall we use traditional C compilation and linking as a first step to obtain the raw material for our templates. While we set the relevant compiler and linker options to generate relocatable code, eventually the *run-time* linker must make some assumptions about the absolute position of code in memory. For example, Sparc `call` instructions code the destination `pc` as an offset relative to the current `pc`. In a given virtual instruction body, we are copying only the `call`, not the instructions at the destination. Therefore, during catenation, the destination field of the relocated `call` must be adjusted appropriately, so that it calls the same absolute address (say, a library routine) as the original `call` in the template. This adjustment is actually deferred until operand specialization time, using the `subst` type `SOT_CALL`.

Similarly, native branch instructions may need fixing after the compiler moves them. However, they are different from calls, in that the majority of branches are local, intra-template control flow. These and their targets are moved together, thus their relative positions are unchanged. We detect intra-template branches that use pc-relative addressing, and do not change them after catenation. Branches with targets outside the template are appropriately repaired.

For an example of catenation, consider the implementation of the `PUSH` opcode code shown in Figure 4.5a. This code uses the indirect threaded code dispatch technique. The C compiler compiles this source to the code shown Figure 4.5b. Now, consider the trivial Tcl bytecode program in Figure 4.5c, to push two objects onto the stack. To interpret bytecode program, the stock virtual machine would execute the instruction body for `push` twice, once for each of the two `push` instructions in the program fragment. Each time, it would also execute the dispatch code to get to the next virtual instruction. Referring to Figure 4.5b, observe that the dispatch code requires eight native instructions, including three memory loads.

Now, recall the key insight motivating catenation. The eight instructions of dispatch code are all virtual machine overhead. The actual work of the `push` opcode, which we refer to as the *instruction body*, is in the first twelve instructions. The catenated version of the program in Figure 4.5c will consist of *two* copies of those twelve instructions, placed consecutively in memory. None of the dispatch code is required, as control flows directly from the first instruction body to the next, just as it flows from Line 1 of Figure 4.5c directly to Line 2.

We should note that the C code we show in Figure 4.5a is using indirect threaded dispatch (recall our detailed comparison of this and other dispatch strategies in Section 2.6.1.) The stock Tcl virtual machine actually uses `for/switch`-type dispatch, which is very similar to indirect threading, in that the C compiler typically implements `switch` efficiently, as a jump table. In practice, most compilers, including ours, `gcc`, compile the `switch` slightly slower than this, adding overhead for a bounds check on the switch selector, and the `for` construct requires an extra unconditional direct jump, to the top of the loop.

Thus, the real dispatch is even worse than we have presented here, and the real improvement from catenation more significant. We compare with the indirect threaded version to show that even carefully crafted interpreter code has overhead. Using the best strategies discussed in Section 2.6, dispatch can be made even faster. However, there is always *some* overhead. Catenation executes fewer instructions than *all* such strategies, because it *eliminates* dispatch.

Operand Specialization Having eliminated all the dispatch code using catenation, we are left with the instruction body. This is the semantic work of an opcode. In the case of `push`, is is the first twelve instructions in Figure 4.5c. Obviously, this work must be performed. But, can it be improved further? Catenation enables further optimizations. The existing code is somewhat generic — it allows for any operand, by fetching the operand from the bytecode stream. Recall that catenation yields a separate copy of the code for *each* virtual instruction. Thus, we can *specialize* the copy using the operand, which is known as a constant. On the first line of our example in Figure 4.5c, the constant operand of the `push` opcode is the index value `0`.

Operand specialization is implemented via interpretation of the `substs` corresponding to the opcode, using the values specified in the operands of the bytecode instruction. In the example, `push`'s operand is an unsigned integer index into the object literal table. Referring to Figure 4.6, note that the operand needs to be placed into the instructions on lines 2 and 3.

The Sparc requires two instructions to set a 32-bit constant, so we have to specialize both. The `subst` will indicate the exact byte offsets of these instructions. Furthermore, the operand specializer will consult the `subst_out_type` field in the `subst` to determine exactly how to undertake the specialization. The native operands of most Sparc instructions are not on exact byte boundaries. In this case, the `sethi/or` instruction pair, a bit mask is used to quickly

```
#define TclGetUInt1AtPtr(p)        ((unsigned int) *(p))
#define Tcl_IncrRefCount(objPtr) ++(objPtr)->refCount
#define NEXT_INSTR                 goto *jumpTable [*pc];

...
case INST_PUSH:
  Tcl_Obj *objectPtr;

  objectPtr = codePtr->objArrayPtr [TclGetUInt1AtPtr (pc + 1)];
  *++tosPtr = objectPtr;    /* top of stack */
  Tcl_IncrRefCount (objectPtr);
  pc += 2;
  NEXT_INSTR;               /* dispatch to next virtual instruction */
```

(a) C-language implementation of `push` opcode

```
add    %l6, 4, %l6           ; increment VM stack pointer
add    %l5, 1, %l5           ; increment pc past opcode. Now at operand
ldub   [%l5], %o0            ; load operand from bytecode stream
ld     [%fp + 0x48], %o2     ; load address of bytecode object from stack frame
ld     [%o2 + 0x4c], %o1     ; load address of literal table from bytecode object
sll    %o0, 2, %o0           ; compute array offset into literal table
ld     [%o1 + %o0], %o1      ; load from literal table
st     %o1, [%l6]            ; store to top of VM stack
ld     [%o1], %o0            ; next 3 instructions increment reference count of pushed object
inc    %o0
st     %o0, [%o1]
inc    %l5                   ; increment pc

sethi  %hi(0x800), %o0       ; the rest is dispatch to next instr
or     %o0, 0x2f0, %o0
ld     [%l7 + %o0], %o1
ldub   [%l5], %o0
sll    %o0, 2, %o0
ld     [%o1 + %o0], %o0
jmp    %o0
nop                          ; branch delay slot could not be profitably filled
```

(b) C compiler's assembly language translation of code in (a)

```
push   0
push   1
```

(c) Sample bytecode program fragment

Figure 4.5: Use and implementation of `push` bytecode instruction. In the text, we show how *catenation* can eliminate dispatch code, and how this subsequently enables other code to be removed, in particular by *specialization*.

```
add   %l6, 4, %l6                 ; increment VM stack pointer
sethi %hi(operand), %o1           ; object to push: specialize operand here (hi 22 bits)
or    %o1, %lo(operand), %o1      ; object to push: specialize operand here (lo 10 bits)
st    %o1, [%l6]                  ; store to top of VM stack
ld    [%o1], %o0                  ; next 3 instructions increment reference count of pushed object
add   %o0, 1, %o0
st    %o0, [%o1]
```

Figure 4.6: Template for `push` opcode, before operand specialization

specialize the values without perturbing the opcode fields of the native instruction.

For `push`, the operand is an index to the object literal table. We could just emit the index value into the compiled code, saving one load (line 3 of Figure 4.5b) from the instruction stream. However, we know *a priori* that the index will always be used to do a lookup in the literal table. We have the table available at specialization time, and can treat it as a constant. We can perform the lookup at compile time, and emit the actual object pointer, instead of its index. This saves us an additional load in the executed code, on line 7, and we can also remove the instruction line 6, which simply computes the address for the elided load. Specializing away loads is particularly profitable, because it reduces memory system traffic.

We actually save two more loads, beyond the two already mentioned, because on lines 5 and 4, the compiler-generated code loads the address of the literal table from the bytecode object, the address of which is stored in the stack frame. So, specialization saves even more than we have discussed already. However, even optimally crafted interpreter code would have to load the operand from the instruction stream, and likely from a literal table, too.

Exploiting catenation, we are able to eliminate still more instructions. The normal interpreter dispatches opcodes by fetching them from the bytecode stream, at the current virtual program counter (`vpc`). After catenation, dispatch is eliminated, so we do not need the `vpc` for this purpose. The normal interpreter also uses the `vpc` to fetch operand values from the bytecode stream. But after operand specialization, operand values are stored in the emitted native code. So, we will not need the `vpc` for this purpose, either. For certain instructions involving exception handling, we may still need to consult `vpc`, see Section 4.1.4.3. However, it *never* needs to be stored, nor, more importantly, updated. Therefere, we can remove the instructions which maintain the virtual `pc`. In our example, these would be the instructions on lines 2 and 12 of Figure 4.5b,

It is possible that a hand-coded interpreter could combine instructions 2 and 3 in Figure 4.5b into a single `ldub [%l5 + 1], %o0` and change the final increment of `vpc` on line 12 to jump over the entire two bytes of the virtual instruction (`add %l5, 2, %l5` instead of `inc`.) Our C compiler, `gcc`, always chooses to separate the increment and the load. This may be due to an instruction scheduling heuristic. In any case, it is a clear advantage of our technique to *never* store or maintain the `vpc`. Even a hand-crafted interpreter cannot avoid dedicating a register to the VM PC, and at least one native increment instruction per opcode body to maintain it.

Returning now to the workings of the specializer, we must indicate to it that it needs to resolve the literal object in `push`'s operand as a run-time constant. We set the `subst_in_type` field of a new `subst`, associated with `push`, to the value `SIT_LITERAL`. Upon encountering the `push1` instruction in a bytecode program, the specializer will consult the `instruction_desc_aux_t` for `push1` to learn it has one single-byte unsigned integer operand. After fetching that operand, the `SIT_LITERAL` will instruct the specializer to translate it through the object literal table. The result is passed on the output stage of specialization.

In this case, the operand needs to be specialized into the `sethi/or` instruction pair in the template, shown on lines 2 and 3 of Figure 4.6. Therefore, the `subst_out_type` is set to `SOT_SETHI/OR`.

Other opcodes will have one of a few other sorts of operands. The various possibilities for each operand type are listed in Figures 4.4. After any transformation directed by the `subst_in_type`, the compiler needs to know exactly how to place the resulting value into a template. In addition to `SOT_SETHI/OR`, seen in our example, there are several other possibilities, which are listed in Figure 4.3.

One special operand type listed in Figure 4.3 is the offset of a virtual jump. We convert these jumps to single native branch instructions. An unconditional jump is converted to a single Sparc `b` (Branch Always) instruction. The conditional jumps are translated to a small block of code which pops the boolean condition value from the virtual machine stack. As with other templates, this code derived from the original interpreter, but the specializer replaces the final manipulation of the virtual `pc` with a single native instruction.

Depending on the type of virtual jump, the native jump instruction is either `bne` (Branch Not Equal) or `be` (Branch Equal). After the native opcode selection, the native destination address is computed using the virtual destination, translated through the offset map computed in the first pass (see Section 4.1.4.1). This translation of virtual jumps to native instructions saves substantial execution time and space, and has particular impact because jumps are very common in bytecode programs.

4.1.4.3 Exception handling

We mentioned above that we do not need to store or maintain the virtual program counter. However, we cannot completely dispense with `vpc`. Any virtual machine system must deal with unexpected behavior, and Tcl's is no exception. Indeed, the Tcl language has rich standard exception handling facilities, and the implementation attempts to unify handling of exceptions in the interpretation process with the language-level exceptions. Furthermore, the language-level exceptions are dynamic, and the target handler for a given exception at a given point in the code may not be known until runtime. Thus, it is looked up in a table, which is maintained by all three entities: the bytecode compiler, the interpreter, and the bytecode program itself. At any time, the table maps ranges of `vpc` values where an exception may occur, called "exception ranges", to a tuple containing destination `vpc` and virtual stack depth, to enable stack unwinding.

We generate native code so that, when an exception occurs, we jump to a native procedure which is generic — there is only a single copy in the whole system. This procedure consults the exception range table cleans up the stack, and jumps to the appropriate exception handling location in the program. If there is no handler, the error is thrown, right out of the currently executing bytecode object, and may be caught by some calling execution thread, either bytecode or C, or may result in interpreter shutdown.

The table lookup needs to know the value of `vpc` where the exception occurred. We pass this as an argument to the generic procedure. Recall, though, that we do not store or maintain the `vpc` during execution. Instead, we synthesize it as needed, from context. Thus, before each call to the generic exception handler, we emit two instructions which set a very short-lived `vpc` register variable, *as a constant.* This is implemented as the `UPDATE_PC` specialization, and is described in Section 4.2.2.

Alternatively, we could have used the native program counter as a key to look up the `vpc` in the offset table computed during the compiler's first pass. However, the scan would be quite slow compared to the present two-instruction implementation.

4.1.4.4 Second-pass Summary

After catenation and operand specialization have compiled all the bytecode to native code in memory, we must take one more step before the code is ready to execute. Modern processors, including the Sparc, have a "Harvard" cache architecture, where the instruction and data caches are split. The JIT compiler emits native code as data, and thus into the D-cache only. Stale copies of the memory may exist in the I-cache, and when we execute the code, it could be drawn from this stale copy. Therefore, after emitting, we invoke Sparc `stbar` and `flush` instructions to empty the processor store buffers and synchronize the I- and D-caches. Finally, then, the code is ready to execute.

In summary, catenation is simple copying of data, with a bit of data transformation along the way to rewrite branches, etc. As such, it runs very quickly. Interpreting the `subst`s for operand specialization is more involved, and yet is still very fast, because most instructions need no specialization, and the rest are handled quickly. The work required is ideally no more than:

1. One load, of the operand from bytecode stream
2. Two indirect branches, one each for `subst` input and output type
3. One load, to get native instruction (or two in case of `sethi/or`) from template
4. Four ALU instructions, to shift and mask operand into template
5. One store, to emit new instruction

This code must be executed for every `subst` of every bytecode instruction in the program being compiled. Most virtual opcodes require exactly one `subst`, although some require zero, two, or more. However, all this is much faster than any compiler requiring symbolic transformations on intermediate code, or any system involving code that is not fixed-sized.

This concludes our discussion of catenation and specialization, and the JIT compiler proper. The compiler core itself is small, and simple. Most of the complexity and computation have been pushed into other parts of the system, which run at system-build time, or only once, during initialization. Therefore, the compiler runs very quickly, and with minimal I-cache footprint.

4.2 Engineering effort

In Section 4.1, above, we described the workings of the core JIT compiler. The compiler catenates and specializes templates, and our presentation assumed these templates were already available. But, where do the templates come from? We address this question now.

The other significant part of our contribution is to show how our simple compiler was crafted by building the templates directly from the existing virtual machine interpreter. We next present the details of that transformation, which is outlined in Figure 4.7. Starting with stock virtual machine interpreter, we will show the several steps taken to coax the C compiler, along with some post-processing of assembly, to produce templates appropriate for catenation and specialization, like that shown previously, in Figure 4.6.

4.2.1 Decomposing interpreter loop into separate instruction cases

In Section 2.6, we described the *Switch*-type dispatch used in the stock Tcl interpreter. To make the interpreter more amenable to change, we first decomposed the cases for each instruction into separate files, so that our build-time tools could process them individually. A single structured file would also suffice, but separate files obviate the need to parse a structured file format.

We performed this decomposition using traditional Unix text-processing facilities, but also manually, using a text editor. An important advantage of our technique is that it adapts well to enhancements to the stock Tcl interpreter. For example, given a new version of the interpreter, new bytecode instructions are simply extracted to new separate files.

In the next two sections, we describe how the individual files are changed, manually and

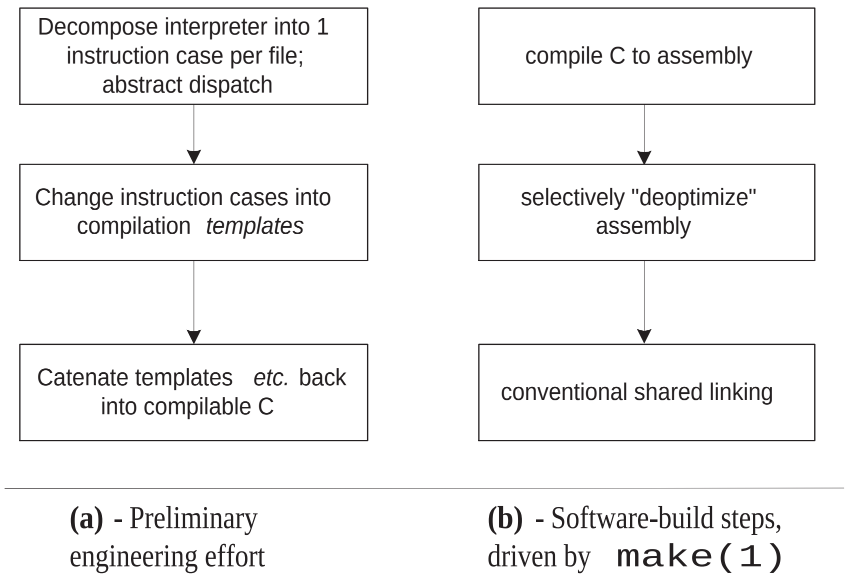

Decompose interpreter into 1 instruction case per file; abstract dispatch
Change instruction cases into compilation *templates*
Catenate templates *etc.* back into compilable C
compile C to assembly
selectively "deoptimize" assembly
conventional shared linking
(a) - Preliminary engineering effort
(b) - Software-build steps, driven by make(1)

Figure 4.7: Overview of engineering effort to synthesize compilation templates from existing interpreter

automatically, to create templates for JIT compilation, and how they are then reassembled into a single large file, ready for the C compiler.

4.2.2 Creating Templates from Stock Instruction Bodies

After the instructions are in files by themselves, we change each to make it more suitable for our work. Initially, to experiment with various dispatch techniques, we removed the dispatch code from the end of each instruction. When re-assembling the cases into a single block of code at build time (see Section 4.2.3, below), we could add back custom dispatch code. This allowed us to test several dispatch strategies, without making manual changes to all 97 opcode cases.

Later, we made other changes to each case, turning them into templates suitable for catenation and specialization. Most of these changes are made by manually abstracting interpreter programming idioms into macros, which could then be redefined to either create an interpreter for testing, or compiler templates to yield the JIT. Aside from dispatch mechanisms, which we discussed in Sections 2.6 and 4.2.1, the main idioms we abstract are for argument handling. This is code that extracts operands from the bytecode instruction stream.

To implement specialization, we redefine this operand-handling code. Continuing with our

```
#ifdef INTERPRET

  #define MAGIC_OP1_U1_LITERAL codePtr->objArrayPtr [TclGetUInt1AtPtr (pc + 1)]
  #define PC_OP(x)                  pc ## x
  #define NEXT_INSTR                break

#elseif COMPILE

  #define MAGIC_OP1_U1_LITERAL (Tcl_Obj *) 0x7bc5c5c1
  #define NEXT_INSTR                goto *jump_range_table [*pc].start
  #define PC_OP(x)                  /* unnecessary */

#endif

...
case INST_PUSH1:
  Tcl_Obj *objectPtr;

  objectPtr = MAGIC_OP1_U1_LITERAL;
  *++tosPtr = objectPtr;    /* top of stack */
  Tcl_IncrRefCount (objectPtr);
  PC_OP (+= 2);
  NEXT_INSTR;               /* dispatch */
```

(a) C-language implementation of `push` opcode

Figure 4.8: Use and implementation of `push` bytecode instruction

example of `push`, our goal is have the C compiler emit the template in Figure 4.6. The code in Figure 4.8 accomplishes this. The `push1` opcode takes a single one-byte unsigned integer operand, which indicates the index of an object in the literal table. (Recall that the instruction's purpose, of course, is to push this object on the interpreter stack.) We replace this code with a C preprocessor macro, which, for specialization, is defined as a unique *magic constant* integer value. As we discuss below, in Section 4.2.4, we arrange for this constant integer value to pass mostly unchanged through all the phases of C compilation, optimization, and linking.

The resulting assembly is exactly the code we need for the JIT compilation *template*. The compiler's initialization phase, described in Section 4.1.4.1, searches for the position and type of the the magic constant. Later, the specializer substitutes actual operand values into that position.

We must specialize the operands of *all* bytecode instructions – the operands are not otherwise available, since we chose to completely compile away the bytecode instruction stream.

`magic_op1_u1`	extract unsigned byte from the bytecode stream
`magic_op1_u1_minus1`	extract unsigned byte from the bytecode stream, subtract one from it.
`magic_op1_u1_literal`	extract unsigned byte from the bytecode stream, translate through literal table
`magic_op1_u4`	extract unsigned word from the bytecode stream
`magic_op1_u4_literal`	extract unsigned word from the bytecode stream, translate through literal table
`magic_op1_u4_builtin_func`	extract unsigned byte from the bytecode stream, translate through table of builtin math functions (e.g. *sin*, *cos*)

Figure 4.9: Magic constants used in interpreter source code to enable specialization in opcode templates

Some operands are just specialized directly, as constants, while others enjoy additional pre-computation treatment as run-time constants, as above with `push`. The various types of operands are listed in Figure 4.9.

By inspecting the assembly output of the compiler, we can confirm which native instructions will be used to load these constants. On a RISC machine, there are only a small number of choices. Constants which fit in a 13-bit signed immediate field are loaded with `mov` instructions (which are actually pseudo-instructions on the Sparc, expanding to `or %g0,` *simm13*`, %rd`; the `%g0` register is always 0.) When we know in advance that our run-time constant will fit in a 13-bit field, we deliberately use a magic constant which will also fit, so the C compiler generates the shortest possible code.

Longer constants always get set with the `sethi/or` idiom. Usually, in the C code we explicitly assign the constant into a variable. In a few places in the interpreter, we use the constant in an expression directly — indexing an array, for example. In these latter cases, `gcc` generates an `add` instruction with the constant in the *simm13* operand field.

Almost every one-byte operand uses `mov`, and every four-byte operand uses `sethi/or`. Recall that some operands are used directly, while others index a table lookup, and the looked-up value is specialized instead. We use different magic values to indicate whether the operand needs translation through a table. In any case, the instruction patterns remain the same; only the magic constant are different. Thus the template scanner can find them all automatically.

A few cases need an `add` instead of a `sethi/or` or `mov`, where the C optimizer applied copy propagation. In a few other cases, the magic number is obscured by a multiplicative or additive constant. For example, the C compiler may optimize address computation for array-indexing code. These constants are stored in `mul` and `add` fields, described above in Section 4.1.2.

This technique is somewhat sensitive to which C compiler we use, and very sensitive to the machine architecture. However, it is much easier than, say, hand-crafting templates. In particular, while we need to learn the instruction patterns used in our templates at specialization points, we do *not* need to know the exact location of these points. The locations are found automatically by the scanning process, at `subst` construction time.

To choose the magic numbers, we collected frequency statistics on one- and two-byte sequences in Sparc programs, to find rare constants. Then, we check for exactly one match when scanning templates (or, in a few cases, which are hard-coded per opcode, exactly two.) The interpreter aborts if it does not find the expected patterns in the template at initialization. While we have never observed one, a collision would immediately be found in testing.

In addition to magic numbers for specialization of operands, we also scan the code for C function calls, and our special `UPDATE_PC` C macro. We generate `SOT_CALL` `subst`s for the calls, indicating that the specializer needs to fix-up the targets of native `call` instructions (see Section 4.1.4.2).

The C code for the `UPDATE_PC` macro expands to an assignment to the interpreter variable `pc`, the virtual program counter. The value assigned is a magic constant. As discussed in Section 4.1.4.3, this value needs to be up-to-date only on certain occasions. When we are catenating the opcode at virtual pc n, we want to specialize the magic constant in any instances of `UPDATE_PC` with n. Thus, when scanning the templates, we locate the assignments, and emit `subst`s instructing the catenater to do just that.

The entire `subst` construction process requires roughly 4 ms on our test machine. This represents only a marginal increase in interpreter startup time (total startup typically lasts 20 ms), because it only occurs once, and is insignificant compared to run-times of even very short scripts. Actually interpreting the `subst`s when compiling a typical bytecode object takes only 120 μs. New bytecode objects are frequently created, and, in our system, always translated to native code. Thus the two-pass system is substantially worthwhile, because recomputing 4 ms worth of work on every translation would be far slower (4000 μs is 3200% overhead for 120 μs of work!)

4.2.3 Build-time creation of interpreter loop

Given the many separate files, each containing C source for the template for one virtual opcode, we need to combine them at build time into a single C function, and output a suitable file of C code. This file contains preamble and epilogue code, including helper functions, C preprocessor `#include` directives, etc. and the function itself, `TclExecuteByteCode()`.

Like the file as a whole, the function itself also contains prologue and epilogue "boilerplate" code. Its prologue consists of the function definition header, including return type, formal parameters, local variables, and some setup code. The setup code is described below. The epilogue contains most of the exception handling code for the interpreter, as well as a small amount of cleanup code which sets the Tcl interpreter run-time "return result" to the last item on the evaluation stack, and finally returns an appropriate interpreter error code (usually `TCL_OK`). Most of this code is derived, or extracted directly from, the original interpreter.

The controlling code of the `TclExecuteByteCode()` function would normally be a `for/switch` loop. Since we will just be catenating the instruction bodies, we don't need any dispatch. We do require, however, exact delineation of the beginning and end of each opcode body. During construction of the function, we insert labels before and after each body. For example, the `mult` opcode gets surrounded by the labels `inst_mult_start` and `inst_mult_end`.

Finally, after the `_end:` label, we place a computed `goto` statement with a variable target:

```
goto *jump_range_table[*pc].start;
```

This gets compiled into an indirect branch, and note that it looks like dispatch code. In fact, it *is* dispatch code, but we never execute it, because during catenation we only use the portion of the opcode body between the `_start` and `_end` labels. This does not include the `goto`, which comes *after* the `_end`. However, the C compiler does not know about catenation. It assumes the code executes sequentially, as written – that control flows from the end of the opcode and into the dispatch.

Indeed, the optimizer has to further assume that the dispatch occurs, and thus that inter-opcode control then flows *non*-sequentially. The array `jump_range_table` contains the addresses of the start of each opcode body. (See Section 4.2.3.) The compiler thus knows that control flows to the start of one of the opcode bodies, but it does not know which. Thus, it is forced to build a control flow graph where each opcode body follows every other. This is precisely what we need, because that is what happens when we catenate code according to an arbitrary bytecode program. In this way, all the register allocation, stack usage, etc., setup by the optimizer for the interpreter, as written, are valid in *any* catenated program.

The `jump_range_table` is an array containing the native addresses of the start and end of the native code for each opcode. Our `build` script places it after the prologue when constructing the `TclExecuteByteCode()` function. In addition to the use just described (i.e., to constrain the control flow graph), this table is also consulted during catenation. Recall that the catenation process consists of copying the native code for each opcode, and so it needs to know where the code begins and ends. In the C code for our compiler, catenation and `TclExecuteByteCode()` are implemented in separate modules, to decompose software complexity. The labels-as-values part of `gcc`'s computed `goto` facility does not allow referring to a label outside of the scope of the function in which it is declared.

To work around this, we extend `TclExecuteByteCode()` to accept an extra argument, `init` (see Figure 4.10.) This argument is a pointer to a structure with the same type as `jump_range_table`. If it is non-null, the function simple copies the `jump_range_table`, which is a local variable, into the array referenced by `init`. At compile time, the compiler must allow for the possibility that `init` will be `NULL`. `TclExecuteByteCode()` is written so that this causes control to flow into the C code for the virtual instruction bodies. In this way, the optimizer cannot see that we never actually call `TclExecuteByteCode()` except to copy the opcode addresses. Otherwise, it could treat the instruction bodies as dead code, and not emit code for them.

Like any typical software built on a Unix system, this process is driven by `make(1)`. It runs the `build` script to create a single `.c` file, and invokes `gcc` to compile the result to assembly. We stop the C compiler at the assembly stage because we interpose a post-processing step before

```
int TclExecuteByteCode(Tcl_Interp *interp, ByteCode *codePtr, execute_export_t *init)
{
  /* ... declare interpreter local variables (e.g. stack, pc) ... */

  static jump_range_table_t jump_range_table []= {
    {&&inst_push1, &&inst_push1_end},
    {&&inst_pop, &&inst_pop_end},
    {&&inst_load_scalar1, &&inst_load_scalar1_end},
    /* ... */
  };

  if (init) {
    init->jump_range_table = jump_range_table;
    return TCL_OK;
  }

  /* ... interpreter setup code; sets up stack, etc. */

  if (! init) {
    goto *codePtr->native_code.code;
  }
  ...
```

Figure 4.10: Beginning of `TclExecuteByteCode()` function

linking. This is described in the next section, 4.2.4. Then, as described in Section 4.2.5, `make` continues to drive the process to complete a traditional build.

4.2.4 Deoptimization of assembly

Given the customized opcode implementations, and building them into a function, as described above, the C optimizer is mostly constrained to build the self-contained opcode bodies we need for templates. However, the opcode bodies are not perfectly suitable, and require some fixing before use. We undertake minor changes to the opcode bodies *after* they are optimized. We do this by post-processing the compiler's assembly output, using a Tcl script which parses the assembly.

The script has minimal knowledge of Sparc assembly language, mostly treating it as text, but recognizing some patterns, such as labels and branch instructions. By identifying and parsing the assembly version of the `jump_range_table`, it is also able to locate the start and end labels for each virtual machine opcode body.

4.2.4.1 Delayed Branch

One optimization that can cause trouble it that of scheduling an instruction to fill the Sparc branch delay slot. The Sparc architecture [23] defines special semantics for "control transfer instructions", including branches. The instruction immediately following a branch is always executed, whether or not the branch is taken. Originally, this was to avoid a pipeline stall while the branch target was fetched. Consider the code for the `store_scalar1` opcode, in Figure 4.11a. It will get compiled into something like Figure 4.11b. Note how the optimizer has inserted a *copy* of the `sethi` instruction at `.L2`, after the branch to `.L3`.

Unfortunately, `.L2` corresponds to the end of our opcode body. The rest of the instructions (including the `sethi`) are dispatch code we do not use. But, the `b .L3` instruction violates one of our constraints for usable opcode bodies: the non-error exit cases must flow to the instruction *immediately* following the end of the opcode body. In this example, that means the exits must flow to `.L2`. Here, the code for the `else` clause indeed flows to `.L2`, but the instructions for the `then` clause end by jumping to `.L3`, one instruction too far. The result, after catenation, is that the first native instruction in the body of the subsequent virtual instruction will be skipped!

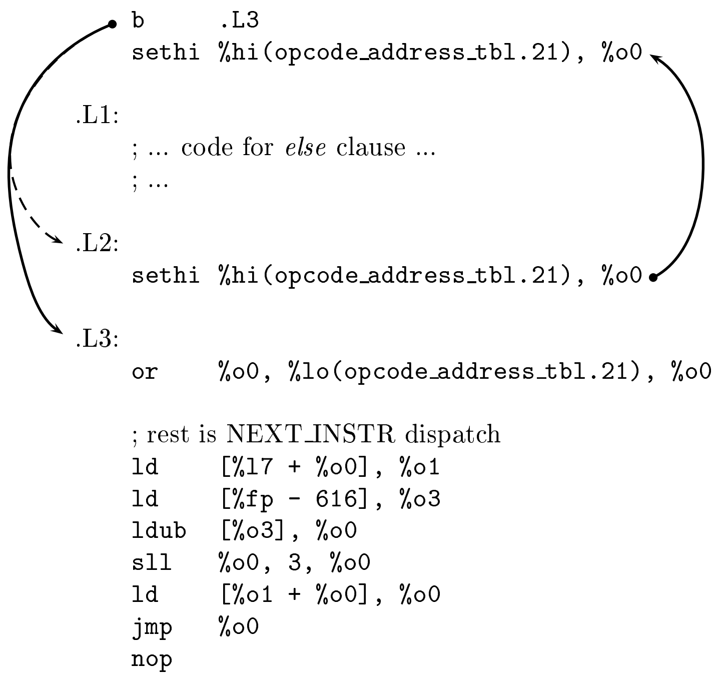
We rectify this problem by reversing the delayed branch optimization, when necessary. Post-processing the assembly, we search for code that matches this pattern:

- A branch B whose target *T2* is exactly one instruction beyond the end of a virtual opcode body.
- The instruction immediately prior to *T2* has label *T1*. Note, *T1* is the end of the virtual opcode body.
- The instruction in the delay slot of (i.e. immediately following) B is the same as the instruction at *T1*.

If we match this pattern, we undo the optimization, by changing the target of B from *T2* to *T1*, and changing the delay slot instruction to `nop`. Fourteen virtual opcodes required this de-optimization.

```
#define NEXT_INSTR \
  goto *opcode_address_table[*pc].start

store_scalar1_start:
  /* ... */
  if (varPtr−>tracePtr == NULL)
    {
      /* then clause ... */
    }
  else
    {
      /* else clause ... */
    }

store_scalar1_end:
  NEXT_INSTR;
```

(a) Excerpt of code for opcode `store_scalar1`



```
      ; ...
      ; ... code for then clause ...

      ; after the then clause, we want to jump to .L2, but the optimizer fills the
      ; branch delay slot with a copy of the instruction at .L2, and changes the
      ; target of the branch to the instruction after .L2, now labeled .L3. The
      ; branch target is now one instruction past the end of the opcode body!

      b      .L3
      sethi  %hi(opcode_address_tbl.21), %o0

.L1:
      ; ... code for else clause ...
      ; ...

.L2:
      sethi  %hi(opcode_address_tbl.21), %o0

.L3:
      or     %o0, %lo(opcode_address_tbl.21), %o0

      ; rest is NEXT_INSTR dispatch
      ld     [%l7 + %o0], %o1
      ld     [%fp - 616], %o3
      ldub   [%o3], %o0
      sll    %o0, 3, %o0
      ld     [%o1 + %o0], %o0
      jmp    %o0
      nop
```

(b) Excerpt of optimized assembly for (a)

Figure 4.11: Code for `store_scalar1` opcode, illustrating de-optimization of delayed branch

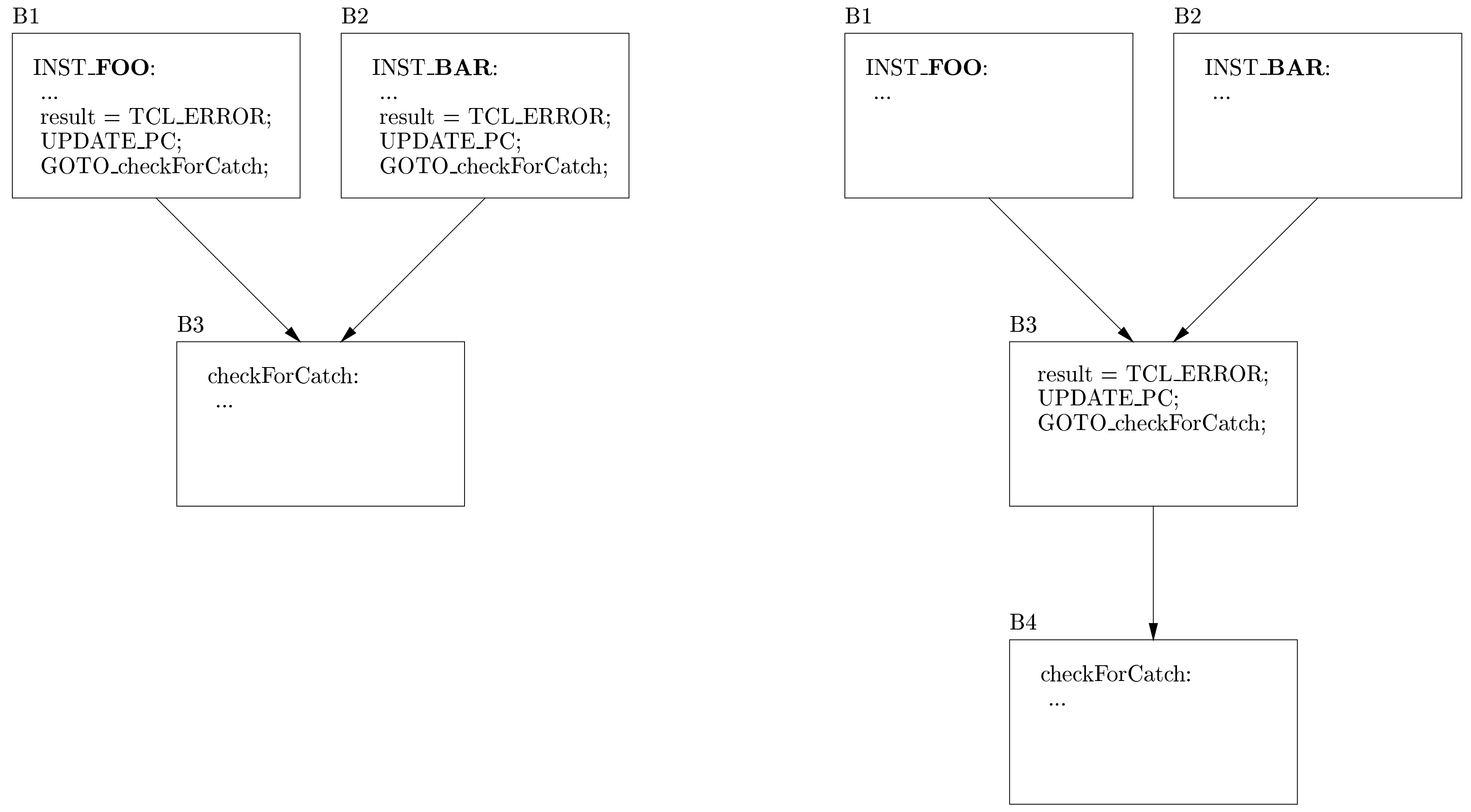
```
inst_foo_start:
  if (value2Ptr == NULL) {
    result = TCL_ERROR;
    UPDATE_PC;
    GOTO_checkForCatch;
  }

  /* ... */

inst_foo_end:
  NEXT_INSTR;

inst_bar_start:

  /* ... */

  if (IS_NAN (R.d) || IS_INF (R.d)) {
    TclExprFloatError (interp, R.d);
    ++tosPtr;
    result = TCL_ERROR;
    UPDATE_PC;
    GOTO_checkForCatch;
  }

  /* ... */

inst_bar_end:
  NEXT_INSTR;
```

Figure 4.12: Multiple blocks of identical code may be abstracted into one when C optimizer applies *tail-merging*

4.2.4.2 Tail Merging

Another `gcc` optimization creates difficulties for us. In undertaking code size reduction, the optimizer employs *tail merging* [27]. Suppose two (or more) nodes in the control flow graph have identical contents, different predecessors, and identical single successor nodes. In this case, the identical nodes may be merged into one, redirecting edges from the predecessor blocks appropriately.

Because many Tcl opcode bodies share similar features to handle dispatch, errors, etc., the optimizer found several opportunities for tail merging in the virtual machine interpreter. This phenomenon is more easily understood in a small example. Figure 4.12 shows implementation of two hypothetical instructions in which this optimization is applicable.



(a) Initial flow graph

(b) Optimized flow graph

Figure 4.13: Problems created by *tail-merging* optimization of C compiler on code in Figure 4.12. The emitted code for at least one of the opcode bodies is no longer contiguous, and thus is difficult to relocate.

Notice that the last three lines of original blocks B1 and B2 are the same, and end in a jump to B3. This pattern allows the compiler to apply tail merging. Unfortunately, the merged code violates our criteria for catenating templates: the opcode bodies are no longer self-contained and contiguous. Instead, one will contain a jump to code stored in the middle of the other. Or, they both might contain a jump to a copy of the code in some other location. Either violates our criteria that all the instructions for the opcode body lie between the `_start` and `_end` labels.

In our example, tail-merging has moved the `UPDATE_PC` macro into a shared block of code in the new B3. Recall that the purpose of this macro is to set a register with the value of the virtual `pc` at a particular instruction. By definition, then, there must be a distinct instance `UPDATE_PC` for each virtual instruction; the instance cannot be shared.

Fortunately, it is not too difficult to transform the assembly language output of the compiler to undo this optimization. Again, a Tcl filtering script treats the problem as a text processing job on the assembly output. It finds the boundaries of all the opcode bodies, loads them all into memory, and parses the labels in the assembly code, so that the target of any branch instruction can be mapped to a given opcode body. Then, it searches for an unconditional branch from one body to another, which in `gcc`'s case, is a signature for this transformation. To undo this optimization, we replace the branch with a copy of the code at its destination.

This concludes our discussion of de-optimization of assembly. It is interesting to note that, after bootstrapping, we have used our catenating Tcl "interpreter" to execute the Tcl scripts used to perform this post-processing. Such self-compilation is an excellent way to find bugs in the compiler.

4.2.5 Linking

After the compiler output has been post-processed for de-optimization, etc., it is ready for assembly with the normal Unix assembler. The resulting object is linked together with all the other object files for the Tcl interpreter, virtual machine, and runtime library. We use a traditional shared library linking process. The resulting library is suitable for embedding in any program, and is also linked with a command-prompt shell and driver program that is suitable for inclusion as the "`#!`" interpreter in the first line of Tcl scripts.

4.2.6 Summary

In this chapter we have presented our implementation of a JIT compiler for the Tcl virtual machine. The compiler is based on the ideas of *catenation* and *specialization*, which use *templates* to generate native code. The templates are fixed-size, and contain almost all the code necessary to implement the semantic body of each virtual opcode. To compile the opcodes in the input bytecode program, the templates are copied in program order to new executable memory. Then, to complete compilation and improve performance, the operands in the program are specialized into slots in the templates.

We derive the templates using a novel approach. Starting with the C code for the original virtual machine interpreter, we use the traditional C compilation process to synthesize the templates. This minimizes effort and semantic errors, and makes it easy for compiler development to track improvements in the interpreter.

A significant proportion of the effort in constructing the compiler is devoted to manipulating the source code and resulting object code, because the C-compiled code is not amenable to arbitrary relocation and manipulation as required by catenation, and the C optimizer can frustrate our strategy of generating templates with slots for run-time constants that can be used by specialization.

Overall, however, these work-arounds are much smaller than the rest of the core compiler, and they allow us to exploit the C compiler to manufacture templates which are high-quality and error-free. Rather than re-implementing the semantics of the existing interpreter, we reuse them. The entire implementation is much smaller, simpler, and faster than a full-blown traditional compilation system. In the next chapter, we describe its performance impact on the run-times of typical Tcl programs.

Chapter 5

Evaluation

To measure the impact of catenation and specialization in our implementation, we constructed two sets of experiments. We report the results of these trials in this chapter.

The first experiment tries to measure execution time on a small number of Tcl benchmarks, to determine if our modified interpreter actually improves performance. It also captures detailed micro-architectural statistics to help learn where these improvements (or, as is sometimes the case, deterioration) in performance originate.

The second experiment tries to answer questions raised by the first, by running a large set of benchmarks on both our Tcl interpreter, and the original version, while varying only the size of the instruction cache. This hypothetical scenario requires a simulation infrastructure, because of the lack of variety in I-cache sizes within generations of the Sparc CPUs. Using CPUs from different generations, which have substantial architectural differences, would confound the results.

5.1 Cycle counts and other Micro-architectural Details

Our first experiment is to measure the execution time of our benchmarks, under stock Tcl and our catenating VM. The highest precision clock available is the CPU's cycle counter, available using the Sparc performance counters [23], which are present in the `sparcv8` instruction set on the Ultrasparc-II and later CPUs. Two 64-bit hardware registers can be programmed to collect any two of a wide variety of events, such as cycles, instructions retired, cache misses, branch mis-predicts, etc. Many of these are of interest, but the most valuable are cycles and instructions. From these we can derive the ratio cycles-per-instruction (CPI), which allow us to measure the throughput of the out-of-order superscalar CPU on our interpreter workload. Thus, in addition to measuring time precisely and accurately, using cycles helps understand *why* our techniques perform as they do.

The throughput ratio is interesting because we know our technique makes tradeoffs in instructions versus cycles. It reduces the number of instructions issued, by completely removing dispatch code, and specializing away some operand processing. The former reduces branches, and both reduce load instructions. On the other hand, we increase the size of the working set dramatically, because of code duplication inherent in catenation. Furthermore, some VM dispatch techniques run afoul of certain micro-architectural features designed to speed execution of traditional, non-VM native programs [9].

To facilitate the experiment, we implemented a Tcl command to collect performance statistics while running arbitrary Tcl code. For any given run, usually consisting of many iterations of a benchmark, we can track two event counters for three virtual machine subsystems: the entire benchmark, all time spent in compilation, and time in catenating / specializing compilation. We can also choose whether or not to include events during execution of system code, on behalf of the application. The events can be selected from most of those available in the hardware, e.g. instructions retired, machine cycles, various stall and cache statistics, etc.. We run many iterations (between 10 and 10000) to reduce the error introduced by measurement overhead and cold caches, and then divide the total elapsed counter by the number of iterations to yield a per-iteration statistic.

For these experiments, we ran our benchmarks on an otherwise unloaded machine, and exclude events incurred while the operating system was executing other programs. The machine is described in Figure 5.1.

The resulting data is shown in the two tables in Figure 5.3, with some details emphasized graphically in Figure 5.4. Each data column is described in detail in Section 5.1.1. See Figure 5.2 for a description of each benchmark program. We chose EXPR-unbraced because it is contrived

Machine	Sun Microsystems SunBlade Model 100
Processor	502 MHz Ultrasparc IIe
Memory	640 MB RAM
Operating System	Solaris 8 4/01
Instruction cache	16 KB 2-way set associative
Data cache	16 KB Direct-mapped
Level 2 cache	Unified 256 KB 4-way set associative

Figure 5.1: Specifications of hardware used in empirical performance evaluation

to force excessive re-compilation, IF-multi-1st-true because it exercises conditional branches, MATRIX-mult-15x15 because matrix multiplication is a common compiler benchmark, MD5-msg-len-10000 and WORDCOUNT-wc2 because each includes a balance of numerical and string processing typical of Tcl programs, and FACT-fact-13 because it is typical of virtual machine interpreter research (e.g., see [33].)

5.1.1 Description of Data Table

Some columns in Figure 5.3 appear twice, once each for `bc` and `ntv`. These correspond to the same statistic measured for both the original bytecode interpreter, and the native code compiler, respectively. A description of each column follows:

Benchmark: Name of the benchmark. Refer to Figure 5.2.

Execute time improve% - cycles: Percentage change in number of native machine cycles executed (all cycles, including all compilation, system, etc.) by benchmark using a bytecode version of the interpreter versus a version compiling the bytecodes to native code. Positive numbers indicate native code is faster. Measured by running benchmark up to 1000 times, and dividing by the number of iterations. Compilation time may thus be amortized in cases where compile result is cached by interpreter.

Execute time improve% - instructions: Same as B, but using machine instructions instead of cycles.

Cycles per instruction: Average number of machine cycles required to execute each native machine instruction.

I-cache stall CPI%: Percentage of machine cycles the CPU spent stalled waiting for machine instructions to be moved from the L2-cache or main memory into the instruction cache.

Load stall CPI%: Percentage of machine cycles the CPU spent stalled waiting for data to be moved from the L2-cache or main memory into the data.

Cycles per VM instruction: Native machine cycles required to execute average virtual machine instruction.

Comp_ntv%: Percentage of execution time spent on compilation, either bytecode or catenating.

IC_hit%: Percentage of I-cache references that hit, when using the catenating compiler.

DC_hit%: Percentage of D-Cache references that hit.

L2_hit%: Percentage of references to the L2 (also known as external) cache.

Sys%: Percentage of all machine cycles executed during the benchmark which were executed by the kernel on behalf of the benchmark.

diff_nocomp% - cycles: Total execution cycles, but not including time spent on catenating compilation.

diff_nocomp% - instructions: Total execution instructions, but not including time spent on catenating compilation.

comp% - ntv: When running the benchmark using the catenating compiler, percentage of compilation time spent catenating alone.

comp% - bc: When running the benchmark using the catenating compiler, percentage of compilation time spent on bytecode compilation (that is, the translation of Tcl source to Tcl bytecodes) only.

EXPR-unbraced	Evaluation of an arithmetic expression using the `expr` primitive. The expression is unbraced, meaning that it must be re-compiled at evaluation time to preserve Tcl semantics
IF-multi-1st-true	A large `if-elseif-elseif...` tree where the first condition is true.
MATRIX-mult-15x15	multiply two 15x15 matrices stored as 2-level nested Tcl lists
MD5-msg-len-10000	Tcl-only implementation of md5 message digest algorithm, run on 10000 byte input
WORDCOUNT-wc2	count words from a file
FACT-fact-13	iterative evaluation of `factorial` function, with input $n = 13$

Figure 5.2: Description of benchmark programs

5.1.2 Discussion of Results

Referring to the data in Figure 5.3a, we see that three out of seven benchmarks run faster when using a VM interpreter using our catenating and specializing compiler. These benchmarks, IF-multi-1st-true, MATRIX-mult-15x15, and FACT-fact-13, are characterized by low average native cycles per dynamic bytecode executed. In general, they are executing loops of simple bytecodes that don't call large bodies of interpreter or system code.

MD5-msg-len-10000 executes substantially fewer machine instructions when using the catenating VM, but this does not translate to reduced execution time, because it executes 46% more machine cycles. Indeed, the catenating VM proves a difficult load for the CPU to execute, requiring 3.46 cycles per machine instruction (CPI), compared to only 1.75 for the normal interpreter. 43% of the CPI is due to the CPU stalling on I-cache misses, which can be seen clearly in Figure 5.4. This is not surprising, since our catenation technique explodes code size by factors of roughly 15, on average. If the resulting working set cannot fit in the CPU's instruction cache, I-cache stalls will overwhelm savings from 20 cycles of reduced dispatch overhead.

In the case of MD5, the inner loop is not tight enough because it invokes a Tcl subroutine — and a relatively large one. Tcl subroutine invocation is relatively inefficient, requiring several C function calls for call frame setup, and a recursive call to the interpreter. These calls, together with the catenated size of the subroutine, exceed the capacity of the I-cache on the Ultrasparc-

IIe.

Except for EXPR-unbraced, the benchmarks all exhibit reduced *instruction* counts. EXPR-unbraced is designed to require continuous late re-compilation. The semantics of Tcl require that `expr` expressions not enclosed in braces cannot be compiled until just before execution. In column K we see this requires over 76% of runtime spent on compilation! It turns out that just over half is native code conversion, and the rest is normal Tcl compilation (see columns T and U.)

Since EXPR spends so much time on compilation, and catenation doubles the length of that time, this and other workloads requiring lots of late compilation are not good candidates for the catenating technique, or any technique which increases compilation time. Unbraced uses of the `expr` primitive are rare, however, and specifically recommended against in Tcl programming style guidelines [18]. Because raw instruction count increase overwhelms the subtler issue of CPI throughput, this benchmark's I-cache performance is not comparable to that of the others.

WORDCOUNT shows an 11% slowdown with catenation. This small degradation seems to have several components. It is likely due to slightly worse I-cache and L2-cache performance (see Figure 5.3b), similar to MD5. WORDCOUNT makes use of system calls to read its input from a file, and is generally more data intensive than most of the other benchmarks. With a large number of cycles for each VM instruction executed, it is not sensitive to interpretation speed, so the small additional compilation time is not repaid.

The WORDCOUNT data load on the L2-cache may interfere with the instruction fetches which missed in the I-cache due to code size increase. In a VM interpreter, the data cache does the job of an I-cache — that is, virtual instructions must be serviced from the D-cache. Thus, we observe interpreter load-stall rates in column I to be quite consistently higher than catenated rates in column H. However, virtual instructions for a bytecode stack machine are compact, and most of the D-cache workload is real data, so this effect is very slight.

Benchmark	Execute time improve%		Cycles per machine instruction		I-cache stall CPI%		Load stall CPI%		Cycles per VM instruction
	cycles	instructions	ntv	bc	ntv	bc	ntv	bc	
EXPR-unbraced	-79	-73	2.10	2.03	15	20	10	15	992
IF-multi-1st-true	13	30	1.84	1.48	21	11	18	19	89
MATRIX-mult-15x15	38	39	1.27	1.25	8	2	23	26	146
MD5-msg-len-10000	-46	26	3.46	1.75	43	12	17	26	269
WORDCOUNT-wc2	-11	4	1.24	1.17	7	5	16	28	2581
FACT-fact-13	61	48	1.21	1.63	6	17	24	17	129

(a)

Benchmark	comp_ntv%	IC_hit%		DC_hit%		L2_hit%		sys%		comp%	
		ntv	bc	ntv	bc	ntv	bc	bc	ntv	ntv	bc
EXPR-unbraced	76	93	88	91	93	50	47	11	16	52	48
IF-multi-1st-true	19	88	96	97	100	52	67	22	9	56	44
MATRIX-mult-15x15	<1	96	99	97	97	68	85	1	1	54	46
MD5-msg-len-10000	<1	77	93	90	91	38	54	2	2	64	36
WORDCOUNT-wc2	1	98	98	97	96	70	74	8	9	51	49
FACT-fact-13	3	97	92	97	95	76	51	5	3	34	66

(b)

Figure 5.3: Cycle counts and other data from running benchmarks with performance counters. See text Section 5.1 for details.

5.2 Varying I-cache Size

Examining the results from the previous section, we see that except in cases contrived to force large amounts of re-compiling of Tcl source (EXPR-unbraced), our technique does decrease the number of instructions executed. However, it often increases the number of cycles executed, and thus overall run-time. A large part of this increase is due to I-cache misses, because catenation dramatically increases the code size of the workload, often beyond the size of the I-cache.

To further explore this effect, we undertook a subsequent experiment, to compare the original interpreter and our catenating version, while running different sizes of I-cache. While all processors in the Ultrasparc II family have a 16 KB I-cache, the Ultrasparc-III processors have 32 KB. We were unable to locate an unloaded Ultrasparc-III, and the architecture is different enough from the Ultrasparc-II to make it difficult to compare only on the basis of I-cache size. Instead, we chose the Simics full-machine simulator [22], which we had also used to collect instruction traces during the development process.

Simics boots and installs an unmodified Solaris operating system from the manufacturer's CD-ROMs. It models several types of Sun machines accurately, including processors, memory subsystems, etc. As such, it is a very faithful simulation platform. Unfortunately, it does not implement the Sparc hardware performance counters used in our earlier experiments.

Furthermore, Simics is a full-machine simulation, and thus makes no distinction between when the simulated operating system has the CPU, and when simulated user processes have it. Thus it cannot directly provide separate accounting for, say, CPU time consumed by one user process of interest, versus time consumed by the operating system on behalf of that process. More importantly, it is difficult to exclude time consumed by another, uninteresting user process which we do not wish to measure.

While we ran the operating system in single-user mode for these experiments, the system still undertakes some activity unrelated to our benchmarks, and thus it is necessary to isolate our statistics collection to only the periods of time when they are running. While this is difficult without operating system source code, we were able to devise an accurate replacement for the per-process cycle and instruction counters, using advice from Simics developers. By

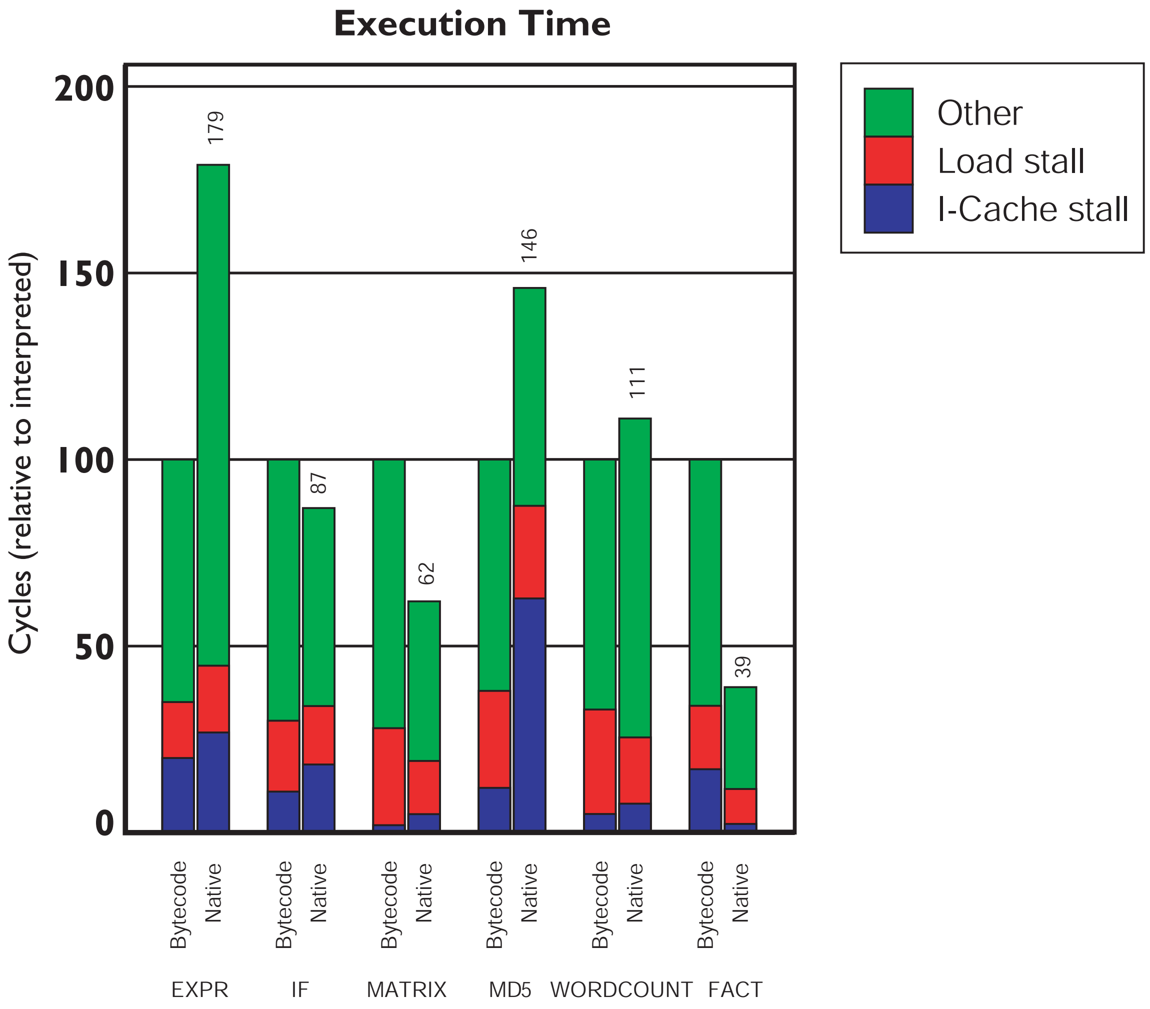


Figure 5.4: Graphical presenation of data in performance counter benchmarks from Figure 5.3.

tracking all "change" events to the simulated MMU context registers, we are able to follow context switches, and thus maintain a per-process data structure accounting for elapsed cycle and instruction counts.

We make this information available to user processes on the simulated machine via a special "magic instruction" interface, which essentially provides an API for communicating between the process and simulator when the user executes a prescribed machine instruction not normally used in the Sparc instruction set. We then encapsulated this API in a Tcl command which can count the number of simulated cycles and instructions executed when running a given number of iterations of a Tcl script. The iteration count and script are specified as arguments to the command.

We ran a large number of benchmarks from the TclBench benchmark suite [41], often used for evaluating the performance of various versions of Tcl. The suite includes many programs typical of Tcl applications, including file and string processing, list manipulation, etc., each executed on a variety of sizes of input.

We configured the simulated Sun Fire 3800 (Serengeti) machine with 512 MB of RAM, an Ultrasparc III+ (Cu Cheetah+) processor, a 64 KB 4-way set associative data cache, and a 1024 KB direct mapped unified L2 cache. We ran four sets of experiments, setting the size of a 4-way set associative instruction cache to 32, 128, 512, and 1024 KB. The L2 cache used a line size of 64 bytes, while both L1 caches had 32 byte lines. Missing in the L1 cache induces a 15 cycle latency, and missing the L2 results in a 100 cycle penalty.

With two Tcl interpreters and four I-cache sizes, each of the 520 benchmarks was run 8 times, resulting in a large amount of data. However, our purposes here require reporting only a few basic numbers (see Figure 5.5.) For the 32 KB I-cache, 46.1% of the benchmarks ran faster with the catenating interpreter. That is, slightly more than half of the benchmarks actually ran slower using our technique. On the other hand, with a simulated 1024 KB I-cache, and a 4 MB L2-cache, 66.3% ran faster. Simulating an infinite size I-cache — a memory hierarchy with no penalty for fetching instructions from main memory — we find that 84.6% of benchmarks run faster. The rest run so quickly that they are unable to amortize the additional compilation time required for native code.

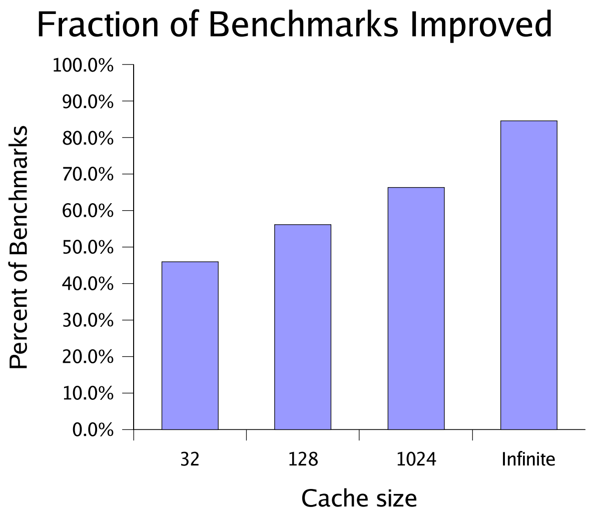


Figure 5.5: Varying simulated I-cache sizes to study code expansion effect versus decreased instruction counts when comparing native-code compiler to original interpreter.

We can see a more detailed picture of how I-cache size affects performance in Figure 5.6. It shows the speedup of all benchmarks for the four cache configurations above. We have *sorted* by speedup the benchmarks *separately per cache size*, and thus the way to read the graph is to see, for example, that for a 32 KB I-cache, 100 benchmarks ran between 16% and 60% faster. For an infinite cache, 200 benchmarks ran between 20% and 70% faster.

The results indicate that larger instruction cache sizes result in *more* improvement in each benchmark for catenation, or, in the cases with slowdown, less slowdown. While this is theoretically interesting, the experiment mainly serves to verify that degraded instruction cache performance due to the code-expanding effect of catenation are the main constraint to its applicability. To be useful, catenation must be applied selectively, using mixed-mode interpretation, or by somehow choosing not to use compilation at all on a given code object. We briefly discuss these ideas in our Future Work section in Chapter 7.

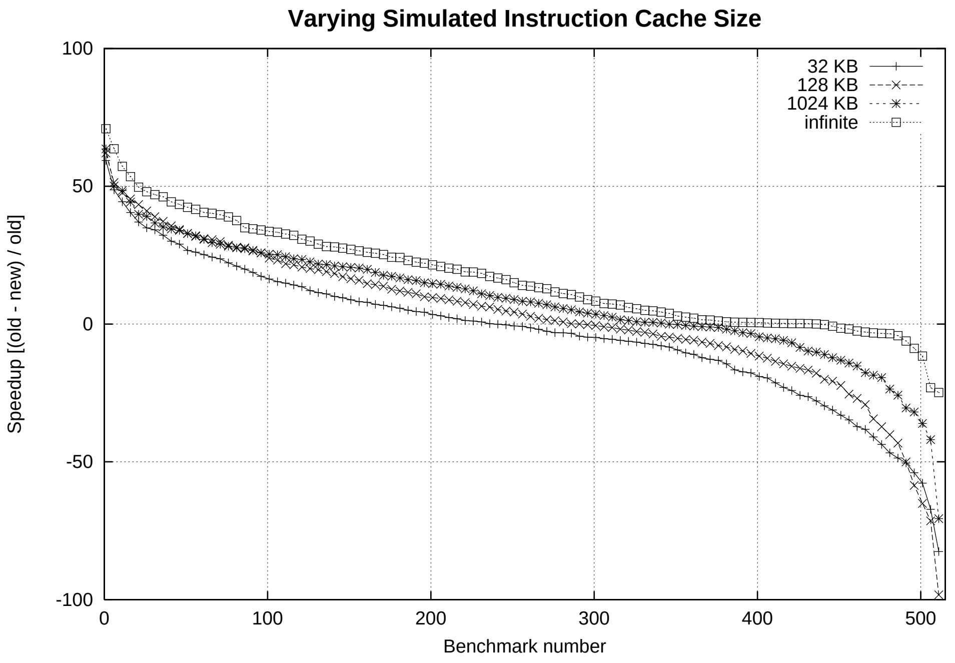


Figure 5.6: Actual speedup of native-code compiler on benchmarks, using varying simulated I-cache sizes. Benchmarks are sorted by speedup.

Chapter 6

Related Work

While compilers have been a well-studied area of Computer Science since they first appear in the early 1950s [1], interpreters have received less attention. Interpreters are used extensively in the practice of programming, but they are often considered inappropriate for performance-sensitive applications, and thus we observe fewer attempts related to improving their performance. In this chapter we discuss some of the research related to our efforts.

Interpreters often appear at the heart of language environments such as Smalltalk, Forth, Java — and Tcl. Together with the large virtual machines in these environments, interpreters facilitate portability, extensibility, and dynamic features such as introspection. Our presentation here is organized partially along the lines of these language systems.

6.1 Forth

Named by its inventor Charles Moore in 1968, the Forth programming language actually started in 1958 as an interpreter of programs containing floating point math primitives [35]. In 1961 it acquired the concept of a stack machine, and later added an additional "return" stack for procedure calls. A significant part of the Forth philosophy includes portability, but with close-to-the-machine implementation and semantics. The Forth interpreters were major early users of indirect- and direct-threaded code.

Anton Ertl [8], investigates various implementation strategies for indirect- and direct-threaded dispatch in a Forth system, and measures their performance on various microprocessors. The data itself is interesting, but most revealing is his conclusion that various modern micro-architectural features have a major impact on the performance of different dispatch strategies.

6.2 Smalltalk

The Smalltalk language began around 1972 at Xerox PARC. Together with its major influence Simula, it is one of the first object-oriented languages [11]. The object system, including garbage collection, and object-based execution paradigm are significantly different from typical hardware, and included features such as animated bit-mapped graphics that were beyond the abilities of that hardware. Thus, the system was implemented as a virtual machine. This machine provided a bytecode interpreter, an object storage system, and *primitive* methods. The rest of the system was implemented in Smalltalk itself, on top of this foundation. This includes the compiler, which translates Smalltalk source methods to bytecode.

Smalltalk is highly interactive, dynamic, introspective, and supports an incremental development environment. Most of the system is closely coupled with the virtual machine interpreter, and so the virtual machine interface had to be maintained in any implementation. Each implementation could run the same Smalltalk *image* (essentially an object database), as long as it closely followed the virtual machine standard specified in The Blue Book [11].

Smalltalk was so useful that substantial applications were built on it, and demand for increased performance grew. Deutsch [6] presents a just-in-time bytecode to native code compilation system for Smalltalk, perhaps one of the first for *any* language. It made relatively naive translations, and allowed for mixed-mode (interpreted bytecode mixed with native) code. It included method inlining, and improved performance substantially.

Today, the highest-performance Smalltalk systems use this native code approach. While the bytecode compiler employs some optimization, there has not been a great deal of activity on improving the interpretation proper of Smalltalk bytecodes. However, the Smalltalk-80 system defined in the Blue Book standard is an evolved version of earlier systems, and includes innovations such as some standard *superinstructions*, which we discussed in Chapter 2.

The Brouhaha Smalltalk virtual machine implementation [26] uses indirect threaded code to interpret standard Smalltalk bytecodes without a compilation phase. The interpreter is constructed using a process similar to our own. The C compiler is exploited at interpreter-build time to generate native code from mostly portable C source. Each bytecode instruction implementation is placed in a separate C function. Assembly language output from the compiler is post-processed to strip prologue and epilogue code. Unlike our system, however, after they are linked and loaded, the instruction implementations are *not* moved, copied, or specialized. Instead, they are used for efficient interpretation, and mechanisms such as the virtual program counter and stack are retained.

Like some other virtual machines, Smalltalk employs special opcodes for *particular* constant values, such as `push_literal`. This opcode takes one operand, an index into a literal table, and pushes onto the evaluation stack the appropriate literal. In addition, it has an entirely separate opcode, `push_1st_literal`, to push the literal table object with index 1. Indeed, it has 16 separate opcodes to push each literal table entry from 0 to 15, and many other bytecodes which pack operand bits into opcode bits. This saves space and time, because the bytecode program can be more compact, and fewer loads are required to decode it (although more bit-level shifting and masking is necessary during decode.) However, it substantially pollutes the bytecode space, which only allows for 256 different entries.

In comparison, our technique allows a more flexible bytecode encoding, with more possible bytecodes, and simpler operand extraction. At the same time, using specialization, it exploits all of the benefits of the above approach using special opcodes, for any operand value. Instead of being limited to efficiently pushing, say, the first 16 literal values, we can compile bytecode which pushes *any* literal value and emit equally dense native code. Furthermore, because specialization is a late compilation, we can propagate the run-time constant value and actual pre-load the literal pointer. So, our technique subsumes customization of bytecodes for certain operands, and executes fewer native instructions in any case.

6.3 Java

Like Smalltalk, Java's object system is integrated in the virtual machine, with a well-defined standard [20]. Due to its widespread popularity, it has attracted a great deal of attention in the compiler research community. Many different interpreter techniques have been deployed to execute Java. For example, the interpreter included in the virtual machine of the GNU Java Compiler, `gcj`, does a pre-pass over the bytecode before execution to translate it to direct-threaded code [3]. Essentially, bytecode opcode numbers are replaced with 32-bit addresses pointing to C switch statement cases with the implementations of each opcode, using `gcc`'s labels-as-values extension.

Many just-in-time compilers (e.g., [43], [39]) have been implemented for Java, which attempt to realize high-performance without changing the bytecode-based software-deployment model, a major part of the Java standard. Some of these, such as HotSpot [24] use advanced run-time dynamic optimization to improve the generated native code. For example, in a form of profile-directed optimization called *polymorphic inline caching*, HotSpot instruments virtual dispatch indirect `call` instructions to learn the class of the receiver object. If there is a strong bias in the distribution of types, the method for the most popular classes can be inlined, after suitable protection with "guard" instructions.

6.4 Tcl

Aside from the bytecode compiler and virtual machine built in to the Tcl interpreter in version 8.0 [18], there have been several attempts at "compiling" Tcl code. The main reasons to compile Tcl scripts are performance, deploy-ability, and protection of intellectual property (source code.) In this section, we discuss some of these efforts.

6.4.1 tc

Perhaps the earliest work is Adam Sah's, which was based on versions of Tcl before version 8.0. For his Master's Thesis [37], Sah[1] invented *dual-ported* objects, which associate a type with

[1]Sah was a student of Ousterhout's at Berkeley.

each string object in the Tcl system, and attempt to cache the value of the typed object. For example, in addition to storing "`123`", the object also stores type `integer` and value `123`.

To see how this improves performance, consider an arithmetic primitive, for example. If only a string value is available for each integer object, it must be parsed into an integer (e.g., using the C `sscanf` function) before the primitive can perform arithmetic, *each time it is used.* Worse, the result must be converted back to a string. With a dual-ported object, the parsed and typed value is cached in the object, along with the string. Primitives can then refer to the value directly, instead of parsing.

Tcl 8.0's object caching system is largely based on Sah's work. Sah also stored a pre-parsed version of scripts, essentially avoiding repeated tokenization. However, `tc` did not attempt any compilation to bytecode or native code.

6.4.2 ICE 2.0 Tcl Compiler

The ICE 2.0 Tcl Compiler project [36] created a commercial stand alone static (ahead-of-time) Tcl version 7 to C compiler in 1995, and then a later version in 1997 that also targeted bytecode and included a conservative type system. The compiler offered an approximately 30% improvement in execution time over the Tcl 8.0 bytecode compiler. Both the ICE compilers were static, that is, required a separate compile step. This precludes using the compiler as a drop-in replacement for the original interactive Tcl interpreter, an important modality for scripting languages. The Tcl 8 compiler, and our interpretation technique, both preserve that modality. The source code of the ICE Compilers was never released to the research community, and is no longer actively developed.

6.4.3 TclPro

TclPro [4] was a commercial software development environment released by Scriptics, a company formed by John Ousterhout, Tcl's inventor, to commercialize the language. TclPro contained a `lint`-like static syntax checking tool, a debugger, an IDE (integrated development environment) and, finally, a "compiler". The compiler's main purpose was intellectual property protection via source code obfuscation, rather than performance. The compiler simply serialized and externalized to disk the Tcl 8 bytecodes resulting from bytecode compilation of scripts. This potentially saves some time by avoiding the compile step at runtime. However, the compiler is so fast that reading the bytecodes from disk takes time comparable to compilation.

6.4.4 kt2c

The Kanga Tcl to C converter [5] is largely intended for intellectual property protection, rather than performance improvement. It is an incomplete prototype, but deserves mention because of its conceptual relationship with our work. Based on the Tcl 8 virtual machine, it translates ahead-of-time Tcl 8 bytecodes to C source code in external files, which is then compiled by a C compiler. The resulting objects have the necessary linkage to be dynamically loaded into a running Tcl interpreter as extensions, but precludes Tcl's interactive mode of work, and complicates deployment.

The C source generated essentially copies the C-level implementation of each bytecode instruction body into the target program, based on the input bytecode object. This is conceptually similar to our catenation process, except it works with source code instead of native. Some operands are specialized into the source, but preserve the Tcl VM literal table, and thus obscuring the value of most constants from the C optimizer. Virtual jumps are converted to native C `goto` statements. The system is incomplete, does not handle all bytecodes, and does not handle exceptions, including `catch`.

6.4.5 s4

A member of the Tcl core development team, Miguel Sofer, created an experimental Tcl VM called `s4` [38]. Among other things, this VM tries to speed up some of the slower but common bytecodes. Some bytecodes require time-consuming calls to helper procedures to handle complex dynamic Tcl semantics, and `s4` tries to inline the common fast cases of these procedures into the instruction bodies in the main interpreter loop. For example, some Tcl variable handling bytecodes must resolve the variable names in hash tables to extract an objects to manipulate the variable. These objects are cached by `s4`, which also inlines the case of a cache hit, so the typical path through the instruction body requires no procedure calls. Rarely-used features such as Tcl's variable "traces", which allow the program to schedule arbitrary code that runs when a variable is read or written, are out-lined to separate procedures.

When compiled with a suitable C compiler, e.g. `gcc`, `s4` also uses indirect threading for dispatch. It also introduces an extra stack to handle reducing the reference count of stack objects consumed during interpretation.

Later, Sofer moved some of the changes into the mainline Tcl interpreter, including caching variable name resolution, and other inlining of the fast typical case of common helper procedures, and indirect threaded dispatch. He also experimented with *wordcode*, which uses 32-bit aligned instruction opcodes and operands, with the goal of eliding the many shift and mask instructions and pipeline penalties necessary when using 8-bit data in RISC memory systems. However, he reported no noticeable improvement here. We observe this 32-bit format could easily support direct threaded code.

6.5 Selective Inlining

In Section 2.6 we defined the concept of *superinstructions*. The limited expressive power of the finite number of bits in a bytecode opcode field means that only a finite number of superinstructions may be defined. Because superinstructions are combinations of two or more instructions, there are potentially a very large number. This means that only a small proportion of potential superinstructions can actually be allowed. Choosing the right superinstructions is thus important, and profile-directed techniques are helpful.

If the virtual machine may be presented with an arbitrary bytecode program to execute, then the optimal set of superinstructions cannot be known in advance, when the machine is built. Piumarta's technique of Selective Inlining [33] addresses this, by building and caching *macro ops* (superinstructions) dynamically, at runtime. In a technique based on GNU C and an existing virtual machine interpreter, he builds one superinstruction, by concatenating copies of the native code for each component instruction into new executable memory.

Note that, as with our approach, concatenating at runtime foregoes the free peephole optimization opportunities when statically building superinstructions at interpreter-build time as in `vmgen` [10].

Piumarta implements his elegant and portable technique in less than 100 lines of GNU C. However, it is severely limited, because it is not able to move code in instruction bodies with any of the following attributes:

- instruction body contains native instructions with native pc-relative semantics, such as most functions calls on RISC processors. When moved, the native instruction will be wrong, because it is pc-relative.
- destinations of VM branches must be left alone, at least in direct threaded code, because the code contains the absolute native address of destination
- flow-control bytecodes that end VM basic blocks must appear only at the end of superinstructions. Piumarta technique manipulates the VM program counter (`vpc`) only once per superinstruction, at the end, as with a normal VM instruction. A flow-control instruction might need to leave a superinstruction early, if it wasn't at the end, and the `vpc` would not be correct.

Our approach does not have any of these limitations. Piumarta benchmarked his techniques in a VM for the OCAML language. He finds that the performance improvements of direct threading and his "selective inlining" can depend on CPU architecture. In one benchmark, computing Fibonacci numbers, a direct threaded VM ran in about 55% of the time of a `for/switch` dispatch on all three architectures tested- Pentium, Sparc, and PowerPC. Relative to direct threaded code, execution time of selective inlined code ranged from 26% to 35% , averaging 29%. The selective inlined code was half as fast as native compiled C.

6.6 DyC

DyC [12] is a selective dynamic compilation system based on annotations in C source that inform the system about run-time constants, which are used in run-time code generation to exploit specialization by partial evaluation. The authors motivate one of their papers using the example of a bytecode interpreter – in this case, `m88ksim`, a simulation of a Motorola 88000

CPU. The input data for this benchmark is a bytecode program. DyC treats the entire bytecode program as a constant, and, using an optimization they call *complete loop unrolling*, is able to essentially accomplish the same effect as our catenation. Because they do this at the level of a control flow graph in intermediate form, they are able to run a subsequent optimization and code emission phase. They still aren't able to do enough automatic optimization to promote virtual machine registers (or stack slots) to native registers. Furthermore, this process is static and quite expensive, and thus might not be appropriate for a dynamic scripting language which frequently compiles and re-compiles. At static compile time, they specialize their run-time specializer so it runs faster and is pre-loaded with most of the analysis for optimization and code generation. This is more general than, but similar to, our `subst` system which pre-computes the necessary fix-ups. However, we interpret the `subst`s while, in a sense, they compile them. They report speedups of 1.8 on `m88ksim`, but do not discuss the complexities of I-cache and code explosion.

Furthermore, in an earlier version [2] of the DyC work, the authors describe a mechanism called the *Stitcher*, an interpreted system which appears similar to our `subst`s in that lists of small data objects guide the specialization of native code.

Chapter 7

Conclusions

We set out to speed up the Tcl interpreter using a simple native code compiler. We believe it is important for this compilation to be very fast, to retain the interactive usage modality of Tcl scripts. We discovered and present a technique on the boundary between interpretation and compilation, which reduces the number of instructions executed and, for some benchmarks, reduces the number of machine cycles, and thus total execution time. In this chapter we summarize our findings, discuss their implications, and briefly outline potential future work.

7.1 Summary

Using the interpreter for the Tcl 8 virtual machine compiled by `gcc` for the Sparc, we found high overhead — about 20 cycles — for the dispatch of each bytecode. As written, this interpreter uses a C `for/switch` construct for dispatch. We believe this is typical of many interpreters coded in C.

Several other techniques for dispatch are available to reduce this overhead. Some well-known approaches, such as "indirect threading" and "direct threading", are possible to implement relatively portably in C using language extensions such as `gcc`'s "computed goto". But because of suboptimal code generation by the C compiler, it may be necessary for a programmer to craft dispatch code directly in assembly. Even then, some overhead is an unavoidable consequence of interpretation. On the other hand, many virtual machines requiring high performance resort to powerful but complex just-in-time compilers which use native code. These are necessarily less portable, and require a potentially time-consuming compilation stage.

We propose new techniques, catenation and operand specialization, to completely eliminate dispatch overhead from an interpreter. The techniques bridge the gap between true compilation and interpretation. The key idea is to use the "instruction bodies" from the actual interpreter as the basis for a simple "template" compiler. We "catenate" these bodies sequentially, as directed by the input bytecode program, to yield native code with no dispatch. Catenation creates a unique code segment for each virtual instruction in the program, instead of the generic implementations an interpreter must use. We exploit this to specialize the operands of the bytecodes into the new native code. This makes the native code smaller, faster, and independent of the input bytecodes, which can be discarded.

In a sense, then, we have "compiled away" the bytecode. However, almost all the runtime infrastructure of the interpreter remains, so it is better to think of this system as an advanced interpretation technique, rather than a true compiler.

Finally, we have devised a technique to create the compilation templates from the original interpreter. We modify the interpreter source to abstract dispatch code, force instruction bodies to be self-contained, and replace operand fetch code with magic numbers. These magic numbers are run-time constants in the resulting templates, and we can substitute real operand values. Because this specialization occurs relatively late in the compilation process, some partial evaluation is possible, such as looking up constant objects and arithmetic functions in indexed tables.

Further post-compile and runtime steps are required to actual implement catenation and operand specialization. These steps are pre-computed at interpreter start-up time for each template. The result is a simple compilation which can run very quickly: conversion from bytecode to native code executes in 50 - 100% of the time required for the first-stage compilation from Tcl source to bytecodes. The original compilation step itself is fast enough as to be imperceptible in an interactive scripting environment, and, crucially, our native code system retains this characteristic.

By eliminating machine instructions for bytecode dispatch, and keeping native compilation overhead low, we succeed in reducing the number of instructions executed by the CPU.

7.2 Implications

On a modern superscalar CPU with a cache memory hierarchy, instruction counts are not the whole story. In our favor, by eliminating dispatch and employing operand specialization, we present the CPU with a workload containing fewer branches and loads — major pipeline hazards. On the other hand, catenation drastically expands code size, which can then easily exceed the capacity of typical instruction caches. Together, these effects can speed up some benchmarks more than 60%. Unfortunately, our measurements indicate the benefits of fewer and simpler instructions are overwhelmed by the latency introduced by I-cache misses in roughly half our benchmarks.

We've shown that our techniques work better given larger I-cache sizes. We conclude that a number of scenarios might make practical use of this compiler:

1. Workloads that fit better in the I-cache, either because the working set is small, the I-cache is large, or both.

2. Machines with simple memory subsystems that do not include instruction caches or penalties for large working sets. While such systems are rare today, some embedded systems may meet these criteria.

In addition to the matter of performance, we also set simplicity, and semantic correctness as goals for our implementation technique. Actually, simplicity is partly implicated by performance, and our fast compilation speed is evidence of the simple design, in addition to the compact implementation. Furthermore, after passing all the nearly 2000 tests in the Tcl test suite, we believe semantic correctness has been achieved, and this is partly due to the reuse of interpreter code.

On the other hand, we believe the technique of reusing interpreter C code, coaxing the compiler to create templates, and massaging these into a suitable form, is not a good platform for general just-in-time compiler research. The semantic information in the C code is not at a sufficiently high-level, and a more abstract "intermediate representation" would be more suitable. Rather than relying on the C compiler, more direct control over code generation is necessary to experiment with different compilation techniques. Infrastructure for intermediate code and code generation will, however, be much larger and slower than our system.

To realize a more practical system, instead of applying our system to narrow application areas such as embedded processors, or discarding it in favor of a more traditional compilation infrastructure, we believe it may be possible to enhance the system, while still retaining the basic strategies of catenation and specialization, to achieve good performance on a wider array of workloads. We discuss this in the next section.

7.3 Future Work

Compared to the interpretive technique of selective inlining, where superinstructions are formed dynamically based on profile feedback (see Chapters 2 and 6), our catenation allows *any* bytecode instruction to be moved. This could only improve the flexibility, and thus the performance, of selective inlining. A synthesis of selective inlining *and* our infrastructure might yield an interpreter superior to either. However, we believe that such a system must still be considered selective inlining, if perhaps a superior implementation thereof.

Another direction found in some JIT compilers is "mixed mode" interpretation. In such a system, bytecodes are sometimes interpreted, and other times compiled and executed native. The result is very similar to selective inlining, but operands are handled more efficiently because of specialization.

Mixed mode could be profitably applied to catenation. With catenation, because nearly all interpreter infrastructure is maintained, switching from interpretation to native execution would be very fast. Theoretically, switching would require only restoral of the virtual program counter (one or two machine instructions to load a constant into a register), and a jump to the interpreter.

Implementation of this idea would require two versions of every instruction body: one for interpretation, and another as a compilation template. Achieving very low switch time would require considerable implementation finesse, because the interpreter state (e.g. execution stack) would have be identical not only at the C level, but also at the register level. In addition to the fast switch time, we would continue to enjoy the high speed of our naive compilation, facilitated by our fixed-size templates and absence of an intermediate form.

To profitably use mixed mode, one must decide *when* to interpret, and when to run native. With catenation, the heuristic is quite clear, since its weakness is excessive code size increase. Therefore, the right approach would be to compile bytecodes with small instruction bodies. These cause the least code bloat, and yet suffer most from dispatch overhead.

The granularity of the mode switch decision could be either entire procedures, or even individual bytecodes. To decide when to switch, the heuristic could use micro-benchmarks to measure I-cache size, and statically aggregated metrics of code expansion as bytecode was compiled. Alternatively, an adaptive approach could be used by exploiting the same hardware performance counters we used in Chapter 5. One could compile and execute native until I-cache miss rates increased, and then bias more toward interpretation.

In conclusion, catenation is a novel interpretation technique with the potential to increase performance of workloads whose working sets fit into the instruction cache after the translation to native code. It is often not practical for programs with larger working sets, but may form the basis of a practical technique when combined with mixed-mode or selective interpretation. While our engineering strategy of exploiting the C compiler for code generation is expedient for implementing catenation, we believe a more traditional code generation infrastructure is appropriate for mixed-mode or other more sophisticated interpreters.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 149–159, 1996.

[3] Per Bothner. A gcc-based Java implementation. In *IEEE Compcon 1997 Proceedings*, pages 174–178, 1997.

[4] Scriptics Corporation. The TclPro Development Kit [online]. September 1998. Available from World Wide Web: `http://www.tcl.tk/software/tclpro/`.

[5] David Cuthbert. The Kanga Tcl to C converter [online]. 2000. Available from World Wide Web: `http://sourceforge.net/projects/kt2c/`.

[6] Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages (POPL 11)*, pages 297–302, 1984.

[7] M. Anton Ertl. *Implementation of Stack-Based Languages on Register Machines.* PhD thesis, Technical University of Vienna, Austria, 1996. Available from World Wide Web: `http://citeseer.nj.nec.com/ertl96implementation.html`.

[8] M. Anton Ertl. Speed of various threading techniques on several processors [online]. 1998. Available from World Wide Web: `http://www.complang.tuwien.ac.at/forth/threading/`.

[9] M. Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. *Lecture Notes in Computer Science*, 2150, 2001.

[10] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. VMgen — a generator of efficient virtual machine interpreters. *Software Practice and Experience*, 32:265–294, 2002.

[11] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1985.

[12] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 1990.

[14] SPARC International Inc. *The SPARC Architecture Manual, Version 8.* 1992.

[15] Neil D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–504, September 1996.

[16] Brian W. Kernighan and Rob Pike. *The Unix Programming Environment.* Prentice Hall, 1984.

[17] Holger M. Kienle. j2s: A SUIF Java compiler. Technical Report TRCS98-18, University of California at Santa Barbara, 1998. Available from World Wide Web: `http://www.cs.ucsb.edu/research/trcs/docs/1998-18.ps`.

[18] Brian Lewis. An on-the-fly bytecode compiler for Tcl. In *Proceedings of the USENIX 1996 Tcl/Tk Workshop*, July 1996.

[19] Don Libes. *Exploring Expect.* O'Reilly and Associates, 1994.

[20] Tim Lindhom and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[21] Mark Lutz. *Programming Python, 2nd ed.* O'Reilly and Associates, 2001.

[22] Peter S. Magnusson and Fredrik Larsson et al. Simics/sun4m: A virtual workstation. In *Proceedings of the Usenix Annual Technical Conference*, pages 119–130, 1998. Available from World Wide Web: `http://citeseer.nj.nec.com/magnusson98simicssunm.html`.

[23] Sun Microelectronics. *UltraSPARC IIi User's Manual.* 1997.

[24] Sun Microsystems. The Java Hotspot virtual machine [online]. 2002. Available from World Wide Web: `http://java.sun.com/products/hotspot/whitepaper.html`.

[25] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[26] Eliot Miranda. Brouhaha - a portable smalltalk interpreter. In *OOPSLA '84 Conference Proceedings*, pages 354–365, 1987.

[27] Steven Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[28] David Orenstein. Open-source programming language finds a home at NBC as it prepares to go digital. *Computerworld*, 1999(2), 1999. Available from World Wide Web: `http://www.computerworld.com/news/1999/story/0,11280,33629,00.html`.

[29] John Ousterhout. Tcl workshop - session summary [online]. 1994. Available from World Wide Web: `http://groups.google.com/groups?selm=2un3k3%24nvg%40agate.berkeley.edu`.

[30] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 1998.

[31] John K. Ousterhout and Jeff Hobbs. History of Tcl [online]. 1999. Available from World Wide Web: `http://www.tcl.tk/advocacy/tclHistory.htm`.

[32] John K. Ousterhout and Jeff Hobbs. Tcl top 10 [online]. 2001. Available from World Wide Web: `http://www.tcl.tk/advocacy/top10.html`.

[33] Ian Piumarta and Fabio Riccardi. Optimizing direct-threaded code by selective inlining. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[34] Lutz Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program. Technical Report 2000-35, Universitat Karlsruhe, Fakultat fur Informatik, 2000. Available from World Wide Web: `citeseer.nj.nec.com/547865.html`.

[35] Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore. The evolution of forth. 28(3), March 1993.

[36] Forest Rouse and Wayne Christopher. A typing system for a multiple-backend tcl compiler. In *Proceedings of the Fifth Annual Tcl/Tk Workshop*, 1997.

[37] Adam Sah. TC: An efficient implementation of the Tcl language. Master's thesis, University of California at Berkeley, 1994. Available from World Wide Web: `http://sunsite.berkeley.edu/TR/UCB:CSD-94-812`.

[38] Miguel Sofer. Tcl Engines [online]. Available from World Wide Web: `http://sourceforge.net/projects/tclengine/`.

[39] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, and Toshiaki Yasue et al. Overview of the ibm java just-in-time compiler. In *IBM Systems Journal*, volume 39, 2000.

[40] Green Mountain Computing Systems. Tcl on Board! embedded platform [online]. 2002. Available from World Wide Web: `http://www.gmvhdl.com/acrodesign/TclOnBoard.ppt`.

[41] The Tcl Core Team. TclLib benchmarks [online]. 2003. Available from World Wide Web: `http://www.tcl.tk/software/tcllib/`.

[42] Larry Wall, Tom Christiansen, and Jon Orwant. Programming perl, 3rd ed. 2000.

[43] Tom Wilkinson. The Kaffe java virtual machine [online]. Available from World Wide Web: `http://www.kaffe.org/`.

[44] Niklaus Wirth. *Algorithms plus Data Structures equals Programs.* Prentice-Hall, 1976.